National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

UNIVERSITY OF ALBERTA

THE RUNTIME ENVIRONMENT AND DEBUGGING IN GUIDE

by

KA SEAN STEPHEN TAM

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL 1990

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-65093-1

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR **KA SEAN STEPHEN TAM**

TITLE OF THESIS **THE RUNTIME ENVIRONMENT AND DEBUGGING IN GUIDE**

DEGREE FOR WHICH THESIS WAS PRESENTED **MASTER OF SCIENCE**

YEAR THIS DEGREE GRANTED **FALL 1990**.

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(SIGNED) .....

PERMANENT ADDRESS:

4613 - 38A Avenue

Edmonton, Alberta

Canada T6L 5B5

DATED 12 OCT. 1990

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled THE RUNTIME ENVIRONMENT AND DEBUGGING IN GUIDE submitted by KA SEAN STEPHEN TAM in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

.....

Supervisor

.....

.....

.....

Date ..... Oct. 10/90 .....

Abstract

The General User Interactive Design Environment (GUIDE) is an interactive programming environment for software development which makes effective use of a graphical user interface. A prototype has been implemented for Modula-2 on a Sun workstation using Smalltalk-80. It provides a set of integrated tools that support program creation, modification, execution and debugging. This thesis describes the runtime enviornment and the debugger view in GUIDE. The debugger view provides a consistent user interface to examine and control the internal runtime structure — a collection of statement trees and activation records. The statement trees are incrementally built as the user is coding the program and the activation records are dynamically allocated at the activation of procedures or functions. Editing capability and incremental compilation within the debugger view allow the user to use it as a single tool for editing, compiling and debugging. In addition, it offers many innovative features such as forward and backward execution, execution history replay and pictorial display of data structures. With its graphical capability, it produces the effects of animation on the flow of execution and in-place updates of data structures. The implementation of many desirable features will help to increase productivity and reduce costs in software development.

Acknowledgments

Praise the Lord! It is His strength and guidance that allow me to keep pressing on and finally finish the task. And I wish to thank my supervisor Dr. Duane Szafron and professor Dr. Jim Hoover for their valuable guidance, patience and encouragement during the preparation of this thesis. Without their support, it would be impossible to complete this thesis. I am also in debt to my wife So-Ling for her understanding and concern during those long hours of sleepless nights.

The prayer supports of many brothers and sisters in Christ are much appreciated. In particular, I wish to thank the Chinese Youth Choir and the Agape Fellowship of Edmonton Chinese Alliance Church, Mr. and Mrs. Edmund and Ivy Ho, and Miss Eva Leung. The strength of their prayers cannot be underestimated.

Table of Contents

Abstract iv

Acknowledgments v

List of Figures viii

1. INTRODUCTION 1
   - 1.1 Importance of Debugging 3
   - 1.2 Problems With Conventional Debuggers 5
   - 1.3 Debugging in Graphical User Interface Environments 6
   - 1.4 Classification of Graphical Systems 7
   - 1.5 Relevant Systems 9
     - 1.5.1 Incense 9
     - 1.5.2 PECAN 11
     - 1.5.3 Dbxtool 13
     - 1.5.4 The PV prototype 13
   - 1.6 Overview of Thesis 14

2. INTRODUCTION TO GUIDE 15
   - 2.1 Introduction 15
   - 2.2 Multiple Views In GUIDE 16
   - 2.3 An Integrated Set of Editors 19
     - 2.3.1 Syntax Directed Editing 20
       - 2.3.1.1 Primitive Structure Editing 21
       - 2.3.1.2 Composite Structure Editing 24
     - 2.3.2 Semantic Error Handling 28
   - 2.4 Summary 31

3. IMPLEMENTATION OF GUIDE 33
   - 3.1 Smalltalk 33
     - 3.1.1 The Advantage 33
     - 3.1.2 The Language 34
   - 3.2 User Interface 37
   - 3.3 GUIDE Architecture 38
     - 3.3.1 GUIDE User Interface 38
     - 3.3.2 Structure Browsers 39
     - 3.3.3 GuideStructure 40
       - 3.3.3.1 StructureList 40
       - 3.3.3.2 GuideParseNode 40
         - 3.3.3.2.1 GuideExpressionNode 41
         - 3.3.3.2.2 GuideEditableNode 41
         - 3.3.3.2.3 GuideCompoundNode 42
         - 3.3.3.2.4 GuideExitNode 43
         - 3.3.3.2.5 BreakPoint 43
       - 3.3.3.3 GuideSymbol 43

3.3.3.3.1 GuideDeclarationSymbol 43
3.3.3.3.2 GuideEnvironmentSymbol 45
3.3.3.3.3 GuideImportSymbol 45
3.3.3.4 SymbolReference 46
3.4 Incremental Compilation 46
3.5 Summary 48

4. RUNTIME ENVIRONMENT IN GUIDE 49
4.1 The Runtime Behavior of Modula-2 49
4.1.1 Activation Record 49
4.1.2 Dynamic Link 50
4.1.3 Static Link 52
4.2 Activation Record 54
4.3 Context Switching 55
4.4 Dynamic Link 56
4.5 Static Link 57
4.6 Storage Allocation 57
4.7 Code Interpretation 59
4.8 Summary 64

5. AN OVERVIEW OF THE GUIDE DEBUGGER 65
5.1 Motivation for a Debugger 65
5.2 Desired Features 66
5.3 The Philosophy of Debugging 72
5.4 The GUIDE Debugger View 74

6. IMPLEMENTATION OF THE GUIDE DEBUGGER 86
6.1 Pictorial Display of Data Structures 86
6.1.1 Display of Primitive Data Structures 87
6.1.2 Display of Composite Data Structures 87
6.1.3 Display of Pointers 89
6.1.4 User-defined Display Routines 92
6.2 Single Stepping of Execution 93
6.3 Execution History 97
6.4 The User Interface of the GUIDE Debugger 99
6.5 Testing of Debugger 101
6.6 Summary 102

7. RECOMMENDATIONS, SUMMARY AND CONCLUSION 103
7.1 Recommendations 103
7.2 Summary and Conclusion 104

References 108

List of Figures

1.1 Software-hardware cost ratio . . . 3
1.2 Cost ratio in software life cycle . . . 5
1.3 Classification of program visualization systems . . . 8
1.4 Default boxed display for the basic types generated by Incense . . . 10
1.5 Two subformats for a record . . . 11
1.6 PECAN display showing execution . . . 12
2.1 Multiple views in GUIDE . . . 17
2.2 Typical screen layout during an editing session . . . 18
2.3 An assignment template in a code view and a dialog window . . . 22
2.4 An in-line dialog of Godel . . . 24
2.5 An IF statement template in a code view . . . 25
2.6 Structure selection . . . 26
2.7 Syntax directed editing using menus . . . 27
2.8 An alert window generated by a semantic error . . . 30
3.1 Instances of class Rectangle . . . 36
3.2 The Model-View-Controller structure . . . 37
3.3 The relationship between the internal structure and the MVC structure . . . 39
3.4 Class hierarchy of GuideParseNode . . . 42
3.5 Class hierarchy of GuideSymbol . . . 44
3.6 Example of a statement tree . . . 47
4.1 Example of procedure calls . . . 51
4.2 Static links for Figure 4.1 . . . 53
4.3 Class hierarchy of GuideStorage . . . 57
4.4 Flow chart of the primitive execution routine . . . 61
5.1 Example of an pictorial display of an integer array . . . 68
5.2 Example of a debugger view . . . 74
5.3 The execution stack pane menu . . . 75
5.4 Expression stepping of anInteger := 4 * (2 + 3) . . . 77
5.5 Example of an inspector view . . . 81
5.6 An inspector view on an array . . . 82
5.7 Display format of data structures . . . 85
5.8 A dialog box for editing a variable . . . 85
6.1 Two display methods of tree . . . 89
6.2 Layout for record containing two pointers . . . 90
6.3 Deep recursive tree showing how elements get smaller . . . 91
6.4 Pointer to data already displayed . . . 91
6.5 Advantage of curved arrows over straight ones . . . 92
6.6 Flow charts of stepping forward functions . . . 94

CHAPTER 1

INTRODUCTION

A graphical user interface (GUI) is becoming a standard feature in modern operating systems [Hayes 89, Seymour 89]. Some older operating systems, such as Unix and DOS, provide a GUI shell while keeping their command line version. Others, such as Macintosh and OS/2, simply have a GUI implemented in their operating systems. GUI has gained popularity over command line systems because it is easier to use and, obviously, more appealing. In many situations, it also increases productivity.

The General User Interactive Design Environment (GUIDE) is an interactive programming environment for use in a GUI environment. A prototype has been implemented for Modula-2 on a Sun workstation using Smalltalk-80. It is intended to support detailed modular design, rapid error-free entry, incremental semantic analysis and interactive testing [Szafron 86b]. Multiple views are used to present a simple consistent user interface to interact with the internal structures representing Modula-2 source code and the execution state of the program. These views are tightly coupled together in that editing in a view often causes explicit changes in other views.

This thesis describes the runtime environment and debugging in GUIDE. The runtime environment consists of the activation and execution of procedures and functions. In this thesis, we will use the term *procedure* collectively to refer to either a procedure or a function, in those cases where the distinction is not important. To simulate the runtime behavior of Modula-2, GUIDE uses two structures: a statement tree and an activation record. The statement tree is the compiled version of the code and is built incrementally

when the user is entering the code. The activation record is dynamically allocated when a procedure is invoked. The object-oriented implementation of GUIDE allows each activation record to control its flow of execution and distributes code interpretation to each node in the statement tree, thus eliminating the need for a centralized code interpreter.

Debugging in GUIDE is done through a debugger view. The debugger view provides a consistent user interface to examine and control the internal state of an executing program. It is well integrated with the rest of the system. For example, deleting a variable declaration in the declaration view would result in an immediate removal of that variable in the debugger view. The objectives of implementing a debugger in GUIDE are:

- to present to the user a single tool to modify, compile and debug a program,
- to implement many desirable features as a debugging tool, and
- to exploit the full capability of GUI.

Typically, programmers use a loosely coupled set of tools to develop software: an editor, compiler, linker, loader and debugger. The GUIDE debugger combines the functions of editing, compiling and debugging into one view. The user can edit and debug the program without leaving the debugger view. Incremental compilation eliminates the need for a separate compiler and the user does not have to control or even initiate the compilation process. The syntax directed editor and incremental semantic analysis ensure the code being entered is free of static errors. The major advantage is that the user is not required to switch back and forth between several different tools and to learn several command languages. Many innovative features are also implemented, such as:

- forward and backward stepping of execution,
- execution history replay,
- immediate execution of procedures without a calling program,

- pictorial display of data structures, and
- hardware independence.

Such features, combined with the effective use of a GUI, will increase programmer productivity and reduce costs in software development.

1.1 Importance of Debugging

The cost of hardware has been decreasing to an affordable range for most home users. A personal computer today costs only about half of what it used to three years ago, yet it is more powerful with more memory and is much faster. Software, on the other hand, is becoming more costly. Figure 1.1 shows the estimate of the software-hardware cost ratio from 1955 to 1985 in the U.S. Air Force; this trend is probably characteristic of other organizations as well.

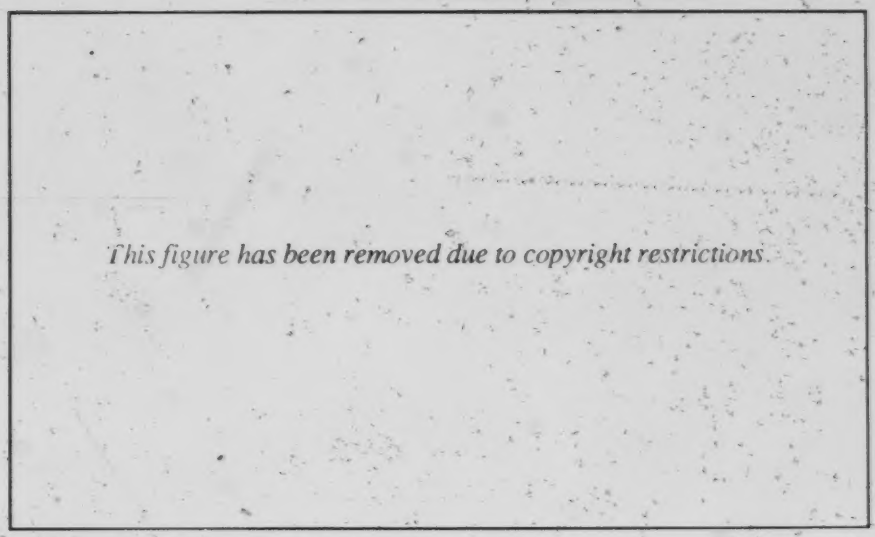


Figure 1.1 Software-hardware cost ratio [Boehm 73].

Software developments are also typically a few years behind hardware technology advances. In other words, software often does not take advantage of the capabilities of new hardware. This is mainly because software is becoming more sophisticated and it takes longer to develop. Even when software is put into production, the first version is usually "buggy" and it takes a few revisions to fix the problems. By the time the bugs are fixed, hardware has already advanced to the next plateau. Software is not easy to develop and to maintain, but a lack of good software development tools has made the matter worse. A good software development tool can help in decreasing both development costs and maintenance costs. If such tools were available, the gap between software developments and hardware technology advances could be narrowed.

In software development, coding is perhaps the most visible activity. Most software developers believe that once a program is coded, testing will be quick and easy. However, experience shows that among the three activities in software development (design, coding and testing), testing is the most costly in terms of time and effort. Typically, design comprises 35% of the development cost, coding 15% and testing 50% [Boehm 73]. Coding is in fact the least expensive step.

Let us consider the entire software cycle — from specification to development and to maintenance. It is a surprise to many people that half of the total cost is spent in maintenance. Figure 1.2 gives an approximate figure on the costs of the various phases in the software life cycle. The high maintenance cost is primarily due to inadequate analysis and design of the resulting software system.

The testing and maintenance phases together consume 70% of the total costs, and they involve a lot of debugging activities. A debugger is an indispensable tool in software

development. Many modern commercial compilers, especially those for personal computers, have realized the importance of debugging by including an integrated source level debugger as part of the environment.

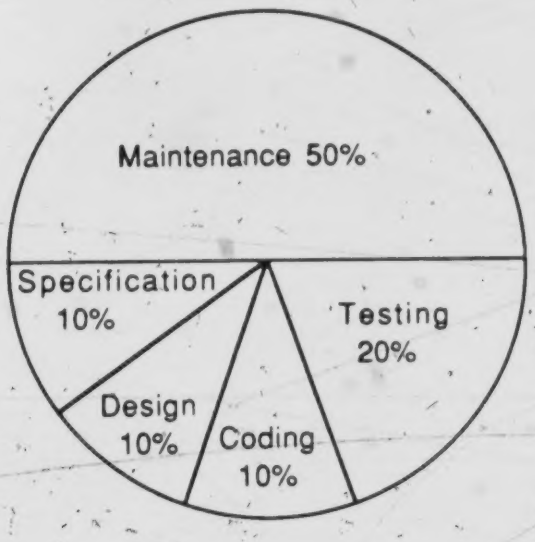


Figure 1.2 Cost ratio in software life cycle.

1.2 Problems With Conventional Debuggers

If debugging is such an important activity, debuggers should be used widely. Yet debuggers, although available, are rarely used. One major obstacle is that most debuggers have complicated command languages that are difficult to learn and remember. Careful design of the command syntax (e.g. dbx [dbx 83]) will help, but on a graphical workstation with a GUI other approaches are possible [Bovey 87]. The complexity is due to the fact that the object under examination, that is the internal state of an executing program, is itself complicated. Furthermore, conventional debuggers are usually hardware dependent. They often present the contents of the internal registers of the underlying machine. But, to a user who is debugging a program written in a high-level language, the

internal registers are irrelevant. Even today, a lot of debuggers (e.g. DOS debug utility [DOS 88], DEBUG program of Tandem Non-stop Systems [Tandem 85]) still present a vast amount of information in a low-level format, such as a hex dump, which is totally incomprehensible to most users.

Because of this, most programmers prefer to use the most straightforward technique of debugging — inserting print statements at various places in their programs. Print statements can be used to display the contents of variables or trace the flow of execution. This method is not without problems. The programmer must first make a conjecture of where the bugs might be, then insert the print statements at the appropriate places, recompile the program, and test the program. This cycle is repeated until the bugs are found and removed. Finally, the print statements must be removed and the program recompiled to produce a final version. Another problem is that the insertion or removal of print statements may modify the behavior of the program in a way that creates or hides bugs.

Conventional debuggers do not meet the needs of today's programmers. They are too difficult to use and often present information at too low a level.

1.3 Debugging in Graphical User Interface Environments

High-resolution graphics is no longer a luxury of expensive workstations as in the past. Technological advances have already brought this luxury down to low cost microcomputers. At the same time, the cost of workstations is decreasing. It will not be long before low cost powerful personal workstations become available and affordable to most people. GUI will become a standard feature on these machines. A graphical

debugger implemented in such an environment will be a valuable tool for software developers.

With the availability of high-resolution graphics and pointing devices such as a mouse, it is possible to design a debugger which provides all the necessary functionality without sacrificing ease of use. A graphical debugger can employ menus and editable subfields instead of a complicated command language for its operations. It can also present information at the user's level — even in the form of pictures. The user will be able to interact with the debugger using the mouse to point at objects of interest rather than typing out their names, thus minimizing mistakes and increasing productivity. More interestingly, the debugger can animate the flow of execution through the source code and dynamically update the variables modified during execution. Users can visualize the animated effect of execution such as the sorting of an array. It would be a pleasant experience for the user to use such a debugger.

1.4 Classification of Graphical Systems

Technological advances of high-resolution graphics and pointing devices has prompted the recent development of systems that utilizes graphics to aid in programming and debugging tasks. The terms "Visual Programming" and "Program Visualization" are often applied to these systems. They provide a completely new way for programmers to interact with software and the programs they write. Moreover, they add the ability to gain different insights and new ways to deal with software through visual and graphical means [Grafton 85]. Myers has defined these terms in a precise manner [Myers 85]:

Visual Programming. "Visual Programming (VP) refers to any system that allows the user to specify a program in a two (or more) dimensional fashion. Conventional textual languages are not considered two dimensional since the compiler or interpreter processes it

as a long, one-dimensional stream. Visual Programming includes conventional flow charts, graphical programming languages, and systems that use icons. It does not include systems that use conventional (linear) programming languages to define pictures."

Program Visualization. "Program Visualization" refers to systems that use graphics to illustrate some aspect of the program or its runtime execution. Program visualization can be classified along two axes: whether they illustrate the code or the data, and whether they are dynamic or static. "Dynamic" refers to systems that animate the running program, whereas "static" is limited to systems that portray the program at some instant of execution. Figure 1.3 classifies some Program Visualization systems according to Myers.

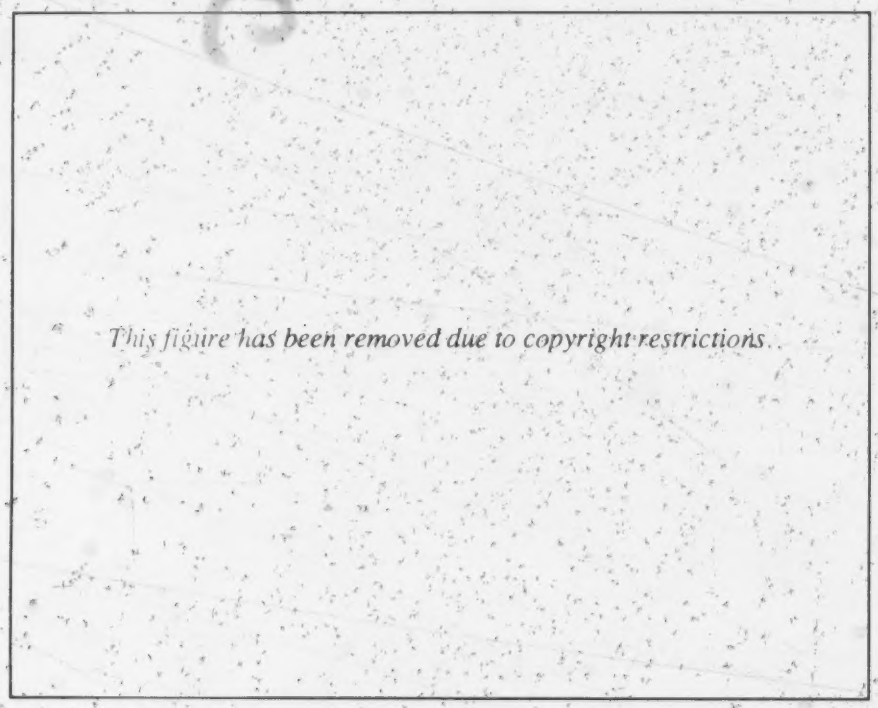


Figure 1.3 Classification of program visualization systems [Myers 85].

A distinction was made: in Visual Programming, the graphics is the program itself, but in Program Visualization, the program is specified in the conventional, textual manner, while some aspects of the program is illustrated using graphics.

Under this classification, the GUIDE debugger is a Program Visualization system, visualizing both code and data. The in-place updates of variables and highlights that move through code to indicate the line being executed make the GUIDE debugger a dynamic system.

1.5 Relevant Systems

The interest in program visualization has aroused the development of many systems [Myers 80, Teitelman 81, Brown 84, Cargill 84 and 85, Delisle 84, Reiss 84, Sherman 84, Brown 85, Teitelman 85, Adams 86, Reiss 86, Sherwood 86, Bovey 87, Isoda 87, Eisenstadt 89, Feldman 89, Guarna 89]. They share a common goal: to aid in program understanding and/or debugging utilizing graphics. Since an exhaust overview of all of them is worthy of a separate paper, we will describe only four of the more well known ones: Incense [Myers 80], PECAN [Reiss 84], dbxtool [Adams 86] and the PV prototype [Brown 85].

1.5.1 Incense

Incense [Myers 80, Myers 83] was designed and implemented at the Xerox Palo Alto Research Center. It was written in and for the Pascal-like language Mesa. The primary goal of Incense was to present data structures to programmers in the way that they would be drawn by hand, thereby making the debugging task easier. Incense automatically generates static pictures for data structures based on the type of the data. Realizing that during debugging, the programmer usually would like to view the data at a higher

conceptual level rather than at a lower level of constituents, Incense allows the programmer to define pictures of his own and modify the displays at various level. These user-defined displays can eliminate unnecessary detail or more graphically portray the abstraction that the data structure is implementing. A set of default displays is provided so that the programmer can be freed from defining any of the types if he chooses to.

Figure 1.4 shows some of the default displays generated by Incense. In addition to displaying data structures, the system supports a number of other operations on the display: erasure, selection and editing.

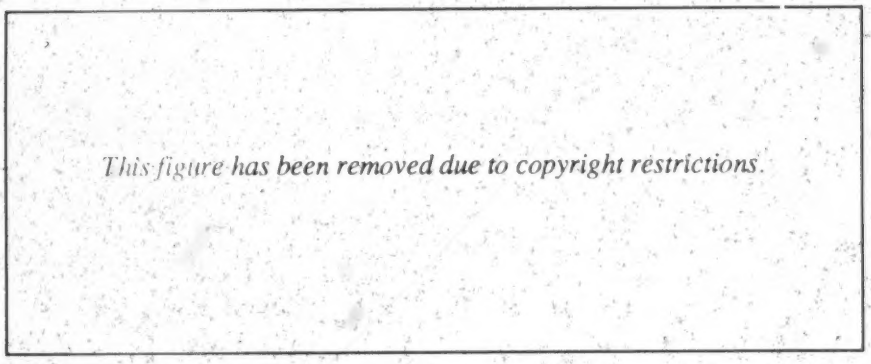


Figure 1.4 Default boxed display for the basic types generated by Incense [Myers 80].

All user input is done using the mouse. To display a data structure, the user uses the mouse to specify the size and position of the display and the system tries to fit the picture into the specified area. When there is insufficient room to display the data at full size, the picture has to be scaled down so that it will fit (Figure 1.5).

Currently Incense does not support any debugging capabilities such as single stepping [Myers 83].

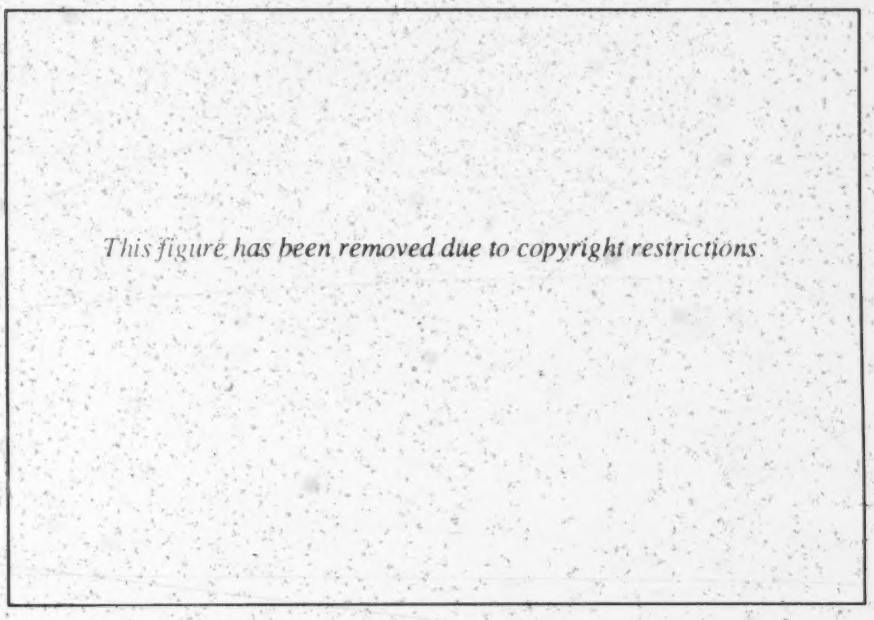


Figure 1.5 Two subformats for a record: full size (a) and scaled proportionally and centered vertically inside a bounding rectangle (b) [Myers 80].

1.5.2 PECAN

PECAN [Reiss 84] is a generator of program development systems for algebraic programming languages developed at Brown University. The PECAN environments provide multiple views to visualize a program's syntax, semantics and executions as the program is being developed. The views supported by PECAN include: program listing, Nassi-Schneiderman diagram, symbol table, data type definitions, parse tree, control flow graph, execution stack, and input-output dialogue (Figure 1.6). An abstract syntax tree is maintained such that if any one of the views modifies the tree, the system automatically updates all other related views to ensure consistency.

PECAN supports interpretive program execution. It allows the program to be executed forwards and backwards, breakpoints to be set, and the speed of execution can be adjusted.

As the program is being executed, a stack view shows a display of the current execution stack. On the stack each activation record is shown and each variable in the activation record is labeled and its value displayed.

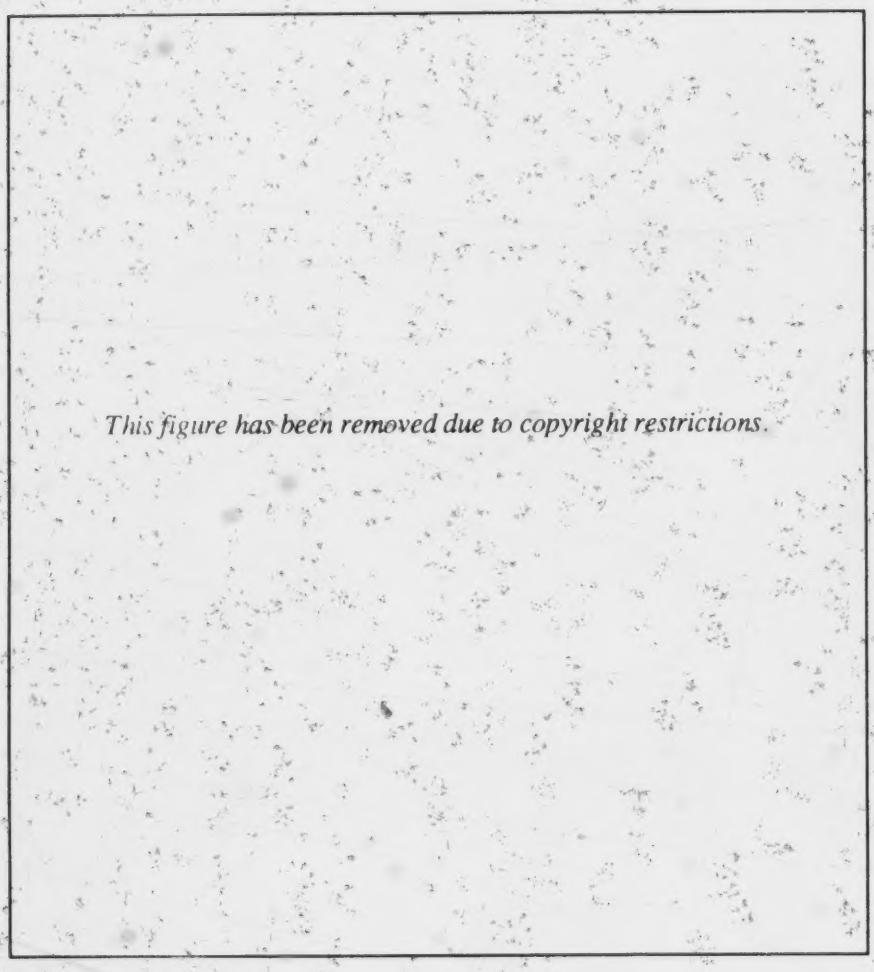


Figure 1.5 PECAN display showing execution [Reiss 84].

1.5.3 Dbxtool

Dbxtool [Adams 86] is a standard software that comes with the Sun-3 workstation. It is in fact a GUI to the teletype-based debugger dbx [dbx 83]. It combines a scrollable editor window displaying the source code with a number of menu buttons representing the most often used dbx commands. The menu commands are object-oriented, i.e. the user first selects the object and then specifies the action to be performed on that object. For example, a breakpoint is inserted by positioning the cursor in the program code and then selecting the **stop at** button from the menu. Similarly the value of a variable of expression is displayed by first selecting it in the code and then selecting the **print** button. However, the use of the menu to examine variable or expression only works when the desired expression occurs in the source code. In order to examine the elements of a more complicated data structure, such as individual array elements or the nodes of a linked list, the user has to type the dbx commands directly into the command window without any help from dbxtool. Dbxtool also makes use of its editor window to display debugging information by annotating the current line and breakpoints with special symbols. Because dbx supports single-line stepping, the user can dynamically visualize code execution through the editor window.

1.5.4 The PV prototype

The PV prototype [Brown 85] provides a set of graphical tools which when tied together can be used to visualize all phases of the software life cycle. The system was written in and for the C language. Related to debugging is its ability to monitor changes in data structures and to show the control flow of a program during execution.

During a debugging session, the user uses a pointing device to point to variables in C code and the system automatically displays appropriate diagrams for the data structure. The user

can also build or select diagrams from a library and "bind" them to the code. Bindings are not implemented by inserting graphic statements into the code, but by establishing correspondences between the code and the graphical components from the information provided by the user. Such a capability allows the user to view the data structures at a higher conceptual level.

Dynamic code visualization is achieved by highlighting the line being executed. The system also provides execution speed control and stepping. If only one data structure is being displayed, the pauses occur only when that data structure is updated. This feature allows the user to localize the debugging to the parts of the program that are of interest.

1.6 Overview of Thesis

GUIDE is an interactive programming environment for software development. This thesis describes the runtime environment and debugging in GUIDE. In this chapter, we have presented a brief introduction to GUIDE's runtime environment and its debugger view. A number of problems found with conventional debuggers can be solved with a graphical debugger implemented in a GUI environment. We have also classified the graphical tools used in software development and presented a few examples. Chapter 2 introduces GUIDE, its user interface and its solutions to many programming environment problems. Chapter 3 describes the implementation of GUIDE and Chapter 4 describes its runtime environment. In chapter 5, an overview of the debugger view is presented. Chapter 6 is a detail description of the implementation of the debugger view. The thesis concludes in Chapter 7 with a summary and recommendations for future work.

CHAPTER 2

INTRODUCTION TO GUIDE

2.1 Introduction

The General User Interactive Design Environment (GUIDE) is an interactive programming environment for the design, coding and testing of software. It is intended to support:

- detailed modular design,
- rapid error-free code entry,
- incremental semantic analysis, and
- interactive testing.

Modular design is accomplished by editing the calling sequence of modules. Rapid code entry is achieved by syntax checking and recovery. Correctness is ensured by eliminating static semantic errors through incremental analysis. As the declarations and code are parsed, they are stored in symbol tables and statement trees. The statement trees can be interactively executed, debugged and tested.

GUIDE is based on an interactive graphical interface which makes extensive use of a high resolution bit-mapped display and a mouse. Besides making the environment much simpler to use, this graphical interface provides solutions to several problems faced by programming environment designers. Some of the problems are: simultaneous representation of multiple views, convenient and regular structure selection and temporary semantic error handling.

The major contributions to programming environments made by GUIDE are its vision of programs as collections of interacting structures and its novel and effective use of a graphical interface. A Modula-2 version of GUIDE has been implemented on a SUN workstation using Smalltalk-80.

2.2 Multiple Views In GUIDE

A program is composed of a collection of modules. A module can be regarded as a collection of statements, a collection of declarations and a header comment which describes the module. Both the statements and the declarations are structured objects whose components may be primitives, like identifiers, or other structured objects. The statements form a tree while the declarations form a list (a special case of a tree). The header comment is the only portion of the module which is represented by a stream of text.

Some of the declarations in a module may be procedures, which are themselves regarded as a collection of statements, a collection of declarations and a header comment. Since the structure of a procedure and a module are so similar, the name *context* will be used to refer to either a procedure or a module, in those cases where the distinction is not important.

The recursive nature of procedure declarations suggests that a context can also be viewed as a tree structure in which the nodes are procedures. Since this tree reflects the static structure of the context, it is referred to as the *static structure tree* of the context.

GUIDE provides the user with a model in which a program is simply a collection of context trees, declaration trees and statement trees. These trees are edited using structure editors. Each of the different editors operates in a view. A view is just a visual representation of one of the aspects of a context. Each view is associated with some internal structure. For

example, the statement editor directly manipulates statement trees and the declaration editor directly manipulates symbol tables.

The current GUIDE prototype supports header, code, declaration, interface and context views. Figure 2.1 lists the five views and the associated internal structures.

View	Internal Structure
• Header	• Text stream
• Code	• Statement tree
• Declaration	• Symbol table
• Interface	• Module: Export list and Import list
	• Procedure: Parameter list
• Context	• Static structure tree

Figure 2.1 Multiple views in GUIDE.

The header view contains the documentation for a context and is edited by a text editor. The other four views are edited by structure editors. The code view contains the statements for a context. The declaration view contains all of the local declarations for a context except procedure declarations. The interface view of a procedure contains the declarations of all of the parameters for that procedure; the interface view of a module contains export and import declarations for that module. The context view of a context contains the procedure declarations which are local to that context. The collection of all context views of a module is a distributed representation of the module's static structure tree.

Each view is presented in its own window and there is no limit to the number of windows which can be opened at one time. For example, the user can look at the declaration view of a context while entering code in its code view. When a procedure call is to be entered, the interface view of the procedure can be opened to determine the required parameters. Figure 2.2 shows a typical screen layout during an editing session.

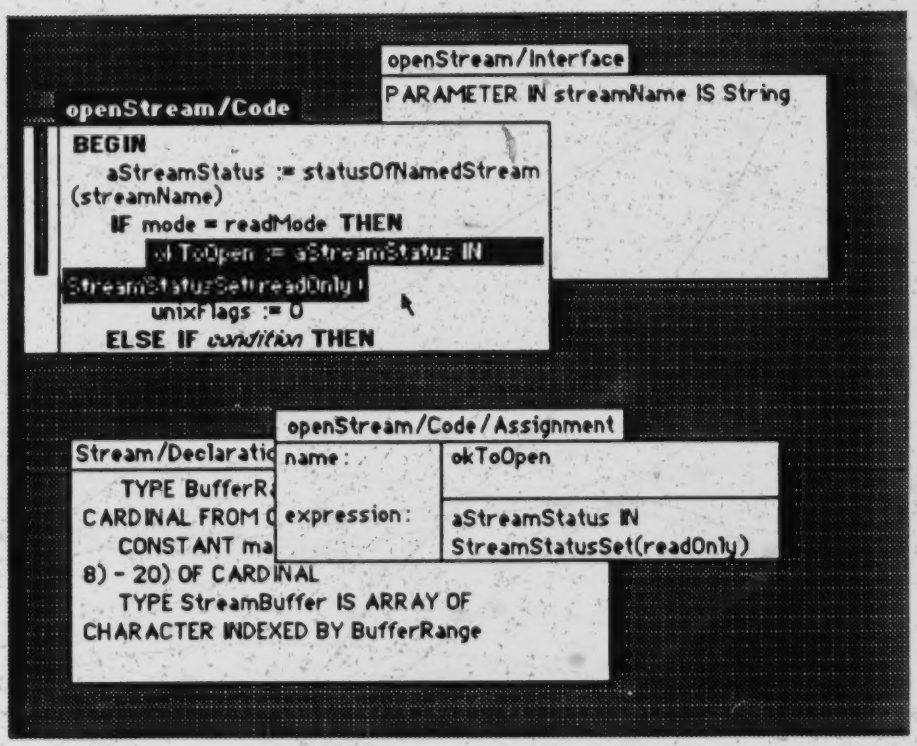


Figure 2.2 Typical screen layout during an editing session.

The removal of procedure declarations from the declaration view offers some distinct advantages. A user interface which presents exactly one way of directly manipulating each

object is easier to learn and understand. This implies that manipulation of procedures (as complete objects) should take place in one view. The question that remains is whether to use the declaration view or a separate view. Note that the manipulation of procedures is inherently more powerful and dangerous than the manipulation of local constant, variable and type declarations, since they encompass so much more information (their own code, declarations and procedures). In fact, the manipulation of procedures involves the manipulation of the program's static structure. Therefore, this difference should be reflected in the interface by providing an orthogonal editing mechanism. A separate context view was chosen instead of a single declaration view to maintain a single regular editing paradigm.

The distribution of context views in representing the static structure tree has one disadvantage: the user has lost a global picture of the static structure of a module. However, this is not as much of a disadvantage as one might expect. In programming languages such as Algol and Pascal, procedure declarations are highly nested so the static structure tree is deep. However, in more modern languages such as Modula-2 and Ada, procedures are usually declared at the global level of modules and the static structure tree of a program is a forest of trees of depth one. In such cases, the distributed context view representation of the static structure does not present much of a problem.

2.3 An Integrated Set of Editors

The structure editor of each view is capable of performing its own syntax directed editing function independent of the other views and their associated internal structures. A common graphical editing paradigm is used in all the views to provide a simple, consistent user interface. In addition, each editor performs static semantic checks as information is entered, incrementally analyzes the information and manipulates its associated internal structure.

While syntax directed editing in each view is independent of the other views, static semantic checks often transcend view boundaries in their communication requirements. For example, when an assignment statement is edited in the code view of a context, the statement editor requires information about the identifier. This information is maintained in a symbol table associated with the declaration view of the context in which the identifier is declared.

In addition to communications between a view and the internal structure of other views, editing in a view often causes explicit changes in other views. For example, changes in the static structure tree must be reflected in the context view of the context containing the declarations of the procedures involved.

Such interaction is the reason why a programming environment cannot be regarded as a set of independent editors. The editors must be integrated in an environment and operate on an integrated internal representation. This integration must not only be efficient, but it must also ensure that editing operations result in changes which are reasonable and predictable by the user. The following presents the solution to this integration problem.

2.3.1 Syntax Directed Editing

A common editing paradigm is used in all of the views. The mouse provides rapid random access to structures of any size and simple regular rules determine the structure being selected. Individual structures can be selected by clicking the left mouse button when the cursor is on the structure. The selected structure is then highlighted. An insertion point can be selected by clicking the left button between structures. A carat is displayed to indicate the location of the insertion point. Once a structure is selected, it can be operated on by selecting an action in a pop-up menu invoked by holding down the middle button when the cursor is inside the view.

The selected structure may be primitive or composite. A composite structure is one which can contain other structures while a primitive structure cannot. Because of this primary difference, they require the use of different editing procedures and syntax checking mechanisms.

2.3.1.1 Primitive Structure Editing

Since a primitive structure cannot contain other structures, it contains a fixed amount of information in a fixed format. Some primitives like the EXIT statement do not require any other information to be specified. An insertion request of such standalone primitives results in an immediate insertion of the primitive. Other primitives require additional information. These primitives when inserted into the view cause a template to be inserted. The information required by the primitives is entered by editing the template. The editing is done by selecting the template and choosing **edit** from the menu which will generate a dialog window for entering the information.

The dialog windows that are used for primitive entry are non-modal. This means that the users are not obliged to fill the box immediately. The user may choose to move the box aside and do something else, and return to the dialog window later. At any time the user can cancel a dialog window by choosing **close** from the dialog window's menu. In this case, the editing operation would be aborted.

All primitive dialog windows used in GUIDE are similar, but the number of fields and the controls may differ. They all have the same menu and a title bar containing the context name, the view name and the primitive structure name.

An assignment statement is an example of a primitive structure. It contains two pieces of information: the variable and the expression. Figure 2.3 shows the template for an

assignment statement inserted in a code view and the window used to enter the information for the assignment statement. A dialog menu which is common to all dialog windows is also shown in the figure. A primitive template is shown in *italics* to differentiate it from the primitive itself.

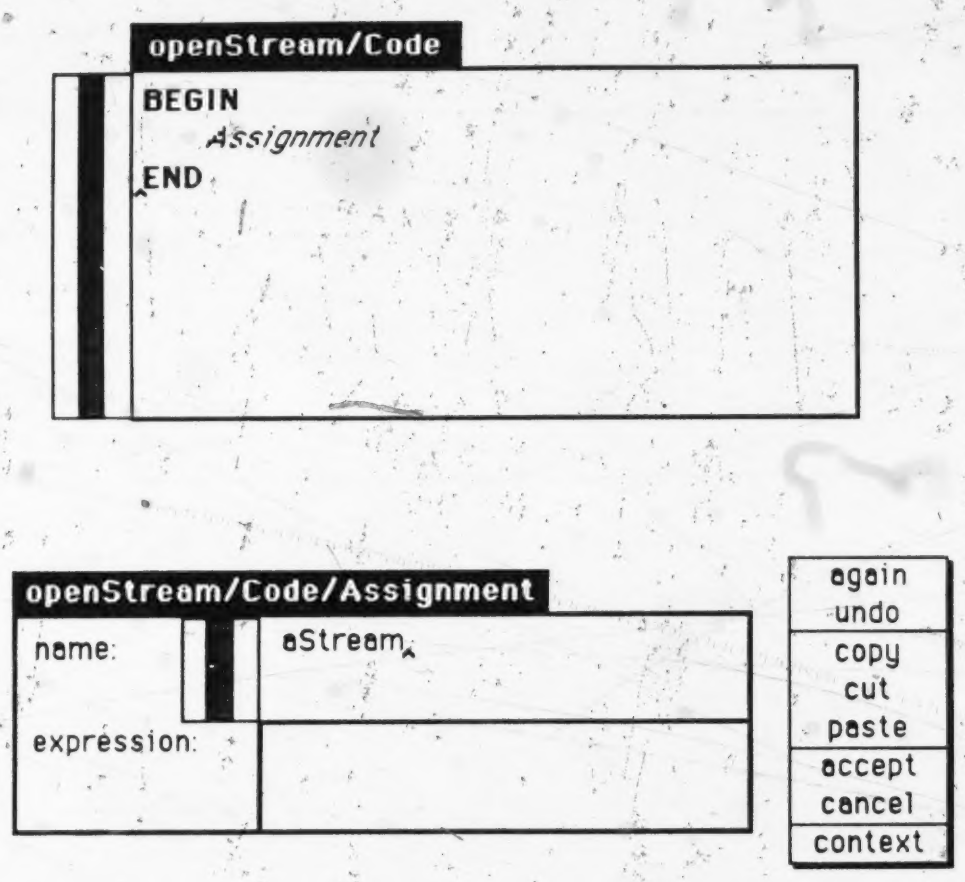


Figure 2.3 An assignment template in a code view and a dialog window.

After filling in all of the fields and selecting the desired controls, the contents of the window may be accepted by choosing **accept** from the menu or pressing the return key

from the keyboard. At this point, the fields are checked for correct syntax. If a syntax error exists, then an error message is inserted into the dialog window, immediately preceding the error and the error text is selected. If no syntax errors are found then the semantic validity is checked. This includes type checking, number of parameters, undefined types, existence of imports, and so on. Section 2.3.2 discusses the handling of semantic errors. If the fields are free of semantic errors, then the template is replaced by the valid primitive structure.

Dialog windows are used not only for editing templates, but for editing existing primitives as well. Primitives cannot be edited directly in the views but the same procedure for editing a template is used. The same dialog window for that primitive is displayed, except that each field will contain its current value. After changing the appropriate field or fields, the **accept** command must be selected from the menu or a carriage return be entered from the keyboard. This results in the same syntactic and static semantic checks that take place during editing a template.

A variant of the dialog window, the in-line dialog, is implemented in Godel [Lanovaz 88]. Godel is a programming environment prototype for Prolog. The design and implementation of Godel are inherited mostly from GUIDE. In Godel, instead of creating a dialog window for text entry, the in-line dialog is embedded within the program clause window (see Figure 2.4). The user selects the position where he wants text entered and then starts typing. The characters entered are surrounded by a thin-lined border. Within this border, conventional text editing is allowed. If accepting the text does not lead to an error, the in-line dialog disappears and the text is automatically reformatted. If errors do occur, the user can make changes and then re-accepting, or excise the text into a conventional dialog window to escape from the in-line dialog mode. If the in-line dialog is

also desired in GUIDE, the Smalltalk source code can be imported from Godel and be integrated easily into GUIDE.

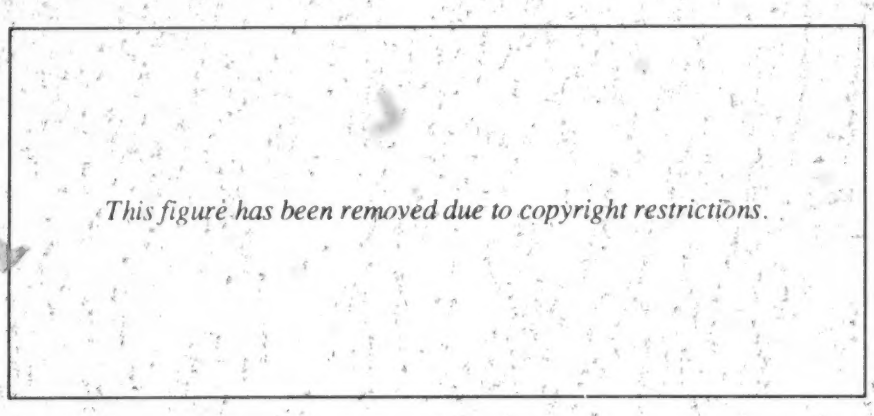


Figure 2.4 An in-line dialog of Godel: the border indicates the in-line dialog start and stop. The insertion carat indicates where typed text is inserted [Lanovaz 88].

2.3.1.2 Composite Structure Editing

An insertion request of a composite structure also results in the immediate insertion of a template into the view. This template can then be filled by inserting other structures into it at a later time. These other structures may be primitive or composite.

An IF statement is an example of a composite structure. It contains a condition and a variable number of statements. The template for an IF statement is shown in Figure 2.5. Notice that the keywords are shown in **bold** so that they can easily be differentiated from identifiers. The null condition is represented by *condition*. Null primitives in composite templates are also displayed using an *italics* typeface. A null primitive can be selected and replaced by a non-null primitive. Of course, a dialog window is used to enter the primitive.

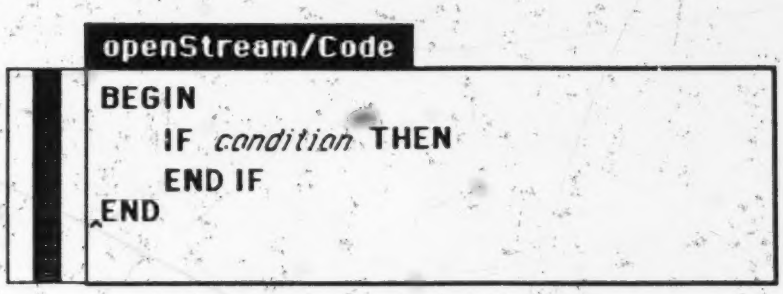


Figure 2.5 An IF statement template in a code view.

Selecting a composite structure is accomplished by clicking the mouse on a keyword that describes the bounds of the composite structure. For example, everything from the **IF** to the **ENDIF** would be highlighted if the user clicked on: **IF**, **THEN**, or **ENDIF**. Clauses are "local" composite structures that can be selected separately from the composite structure that they lie within. An **ELSE IF** is a clause local to an **IF** statement. Clicking on a boundary keyword of a clause will select everything from the first clause keyword to the end of the last statement that the clause holds. A selected composite structure can be cut, copied, or deleted, but it cannot be edited.

Multiple structures can be selected by dragging the cursor across them while holding down the left mouse button. The following rules govern which structure is selected:

Rule 1. Dragging across any two structures at the same level of indentation selects all structures at that level.

Rule 2. Dragging across structures at different indentation levels selects the outermost structure touched and all structures contained within it.

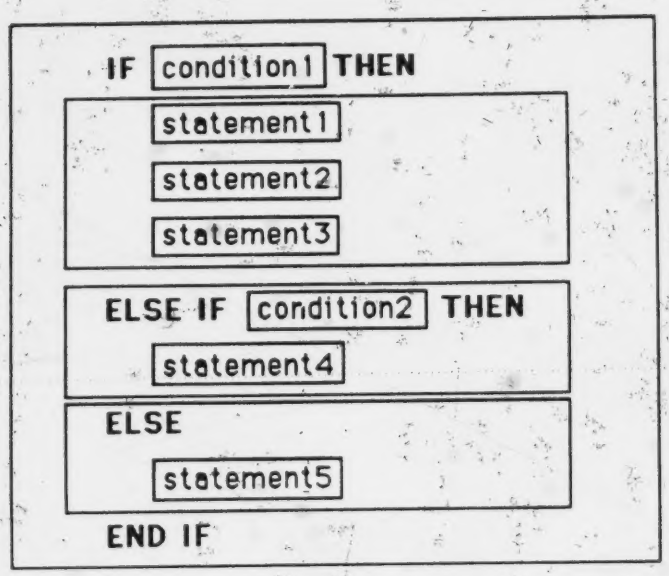


Figure 2.6 Structure selection.

For example, suppose an **IF** statement contained three statements and two clauses as in Figure 2.6. Dragging across statements 1 and 2 would select the block containing statements 1, 2 and 3. Dragging across statements 3 and 4 would select the entire **IF** statement. Dragging across condition 2 and statement 4 would select the **ELSE IF** clause.

Syntax directed editing is done by displaying only allowable items in the menu. For example, if the condition expression of an **IF** statement is selected, then the menu shown in Figure 2.7 is displayed which allows only **again**, **undo**, **copy**, **cut** and **edit** to operate on the condition expression; if an insertion point is selected within the **IF** statement, a hierarchical menu (Figure 2.7) with **again**, **undo**, and **add statement** is displayed. Selecting the **add statement** operation will display a statement sub-menu which shows all the statements that are allowed to be inserted at that insertion point. Users are not allowed

to type text directly into the view but must edit by choosing items from the menu. In this way, a simple but effective syntax directed editing paradigm is achieved.

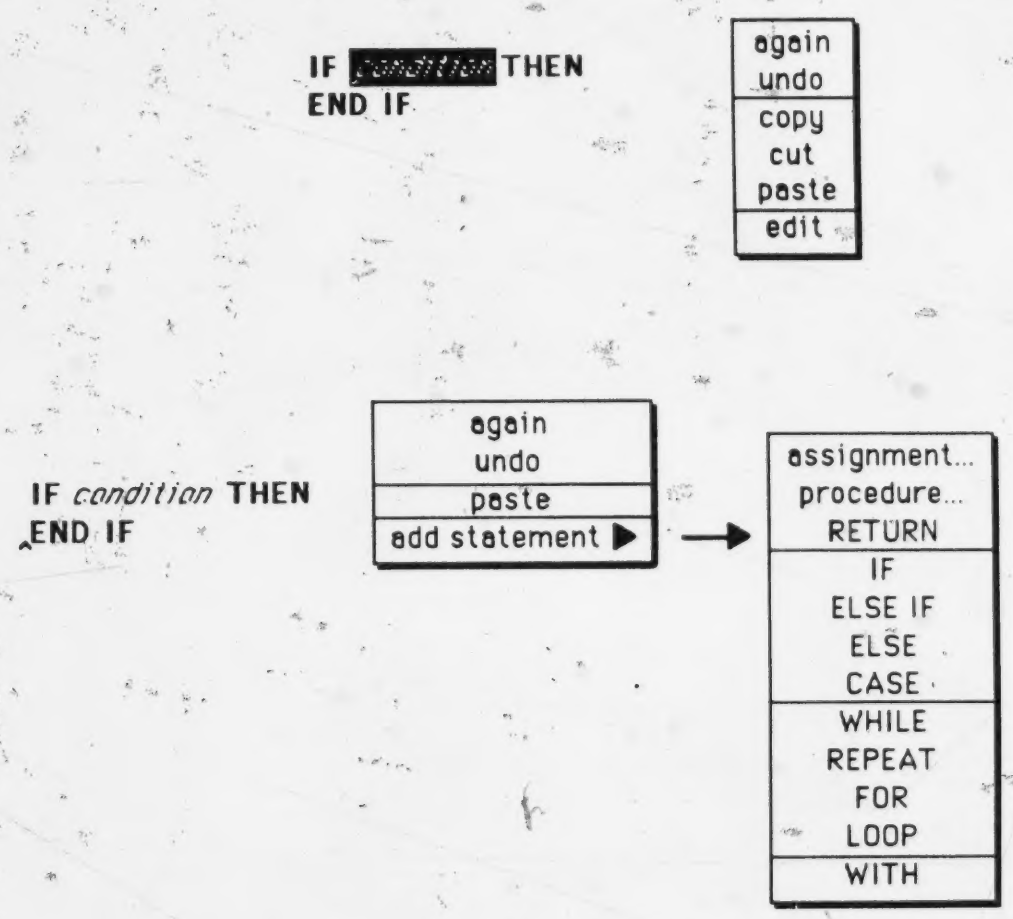


Figure 2.7 Syntax directed editing using menus.

2.3.2 Semantic Error Handling

One of the major decisions which programming environment designers must make is whether to allow semantic errors on a temporary basis. For example, if the user enters an assignment statement which contains an undeclared variable, should the statement be rejected or should it be accepted and flagged. If the statement is rejected, the internal structure of the program is consistent but the user must re-type the entire statement. If the statement is accepted, then the environment must track a sequence of inconsistencies which may become quite complex.

This complexity is due to the wide ranging consequences of simple editing operations. Consider the situation where a variable declaration is deleted. All references to that variable become invalid. Removing all of these references from the code results in a major hardship for the user who must re-enter all of the statements involving these references.

On the other hand, maintaining semantically incorrect software defeats the purpose of a programming environment in two ways. The first is that a programming environment should prevent the user from making subtle errors, much the way strong typing works in a programming language. The convoluted inconsistencies that can arise from a series of editing operations can introduce subtle errors.

For example, consider the situation where a user makes many references to an undeclared variable, say Stack. Of course they are flagged as references to an undeclared variable, but the user plans to fix this later. The user then inadvertently declares the name Stack to be a type and subsequently makes many references to this type in variable declarations. Finally, the user decides to execute some code and is faced with a warning: "references to the name "Stack" which should be declared as a variable, but is declared as a type". Unfortunately, changing the declaration from a type declaration to a variable declaration will just result in

another warning: "references to the name "Stack" which should be declared as a type, but is declared as a variable." Of course this problem can be fixed by adding another name and changing some references, but it is tedious.

The second way that allowing semantic inconsistencies defeats the purpose of a programming environment is that it prevents code from being immediately executable. Programming environments distribute the semantic analysis over time so that users can execute the software at any time. This is crucial to fast software development. If a collection of inconsistencies is maintained, then the user must be notified of these flaws and correct them before execution can proceed. This can be a tedious process.

In fact, the decision as to whether or not to allow semantic inconsistencies is probably the single factor which most affects the usability of a programming environment. GUIDE's graphical user interface provides a solution with the advantages of both approaches and the disadvantages of neither. Only semantically correct information is allowed in the views, but incorrect information does not have to be re-typed since it is saved in the dialog window while measures are taken to correct the semantic error.

When a semantic error occurs in GUIDE, an alert window is generated. It contains a description of the cause of the error. For example, Figure 2.8 shows an alert window generated by a code view when the user tries to enter a control expression containing an undeclared variable in a **FOR** statement.

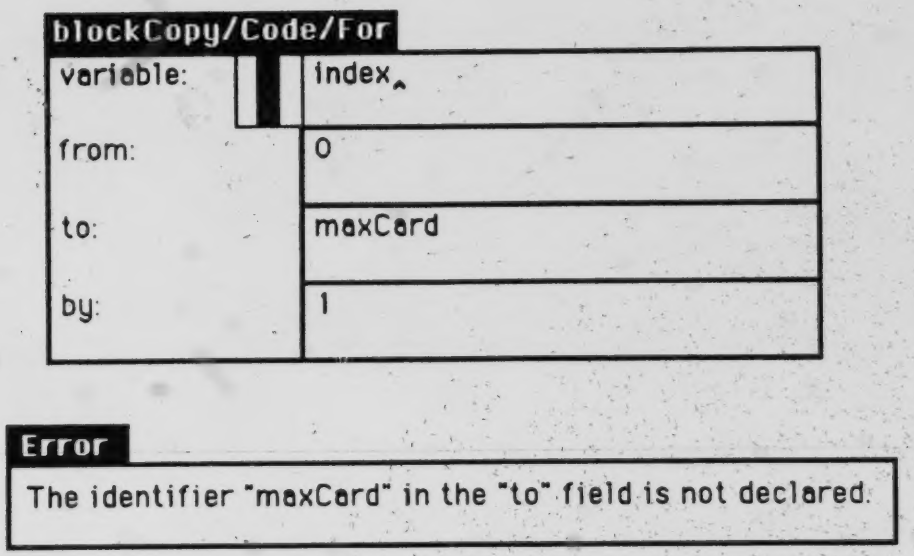


Figure 2.8 An alert window generated by a semantic error.

Alert windows are modal, that is, the user cannot perform any action other than reading the information and closing the window by pressing the carriage return key or selecting the single command **close** from a pop-up menu. Moving the mouse outside of the alert window causes the window to flash. All other user actions are ignored.

The contents of the dialog window are unaffected by the alert. After closing the alert window, the user can edit the fields of the dialog window and then try to re-accept it, or he can **close** it. Alternately, the user may wish to put the dialog window aside and edit another view in order to correct the cause of the semantic error. In this case, the user can return to the dialog window later. There is no need to re-type the information in the dialog window. When the user returns, the **context** command in the dialog window menu becomes useful. Choosing this command displays the window and highlights the template or the original primitive structure which the new structure will replace if accepted.

If no errors are detected, the new structure replaces the template or the original structure and the dialog window is removed. The information in the views is always semantically correct and consistent.

If the primitive structure that was deleted or modified was the declaration of an identifier, then this may result in semantic errors where the identifier is referenced. The interface, context and declaration views of a context all contain declarations. The interface view of a procedure contains parameter declarations, the context view contains procedure declarations and the declaration view contains local constant, type and variable declarations.

In the case of a name change to an identifier, GUIDE avoids the generation of semantic errors by changing all references to the old identifier name to references to the new identifier name.

Semantic errors can be generated if an attribute of an identifier (such as the type of a variable) is changed or an identifier is deleted. If a change in a view causes semantic errors, the user is informed by an alert window and asked if the change should be made. If the change is made, then all inconsistent structures are excised from their views, and an entry dialog window is created for each one and the windows are stacked in a corner of the screen. In this way, the user is free to make other changes which will make the structure correct (for example, by importing a declaration from another module). Then, each structure can be accepted into the appropriate view without re-typing it.

2.4 Summary

The General User Interactive Design Environment (GUIDE) is an interactive programming environment for the design, coding and testing of software. Its major contributions to programming environment research are its successful vision of programs as collections of

interacting structures and its novel and effective use of a graphical user interface. It demonstrates that a software development environment can enforce semantic consistency while maintaining a simple regular user interface. It is also one of the few programming environments in which the user can concentrate on programming instead of concentrating on learning an awkward user interface which includes an unintuitive collection of key presses, modes, rules and exceptions.

CHAPTER 3

IMPLEMENTATION OF GUIDE

This chapter describes the implementation of GUIDE. We first introduce Smalltalk, which is the implementation language for the prototype, and the concepts of object-oriented programming. We then describe the user-interface paradigm used by Smalltalk and the architecture of GUIDE. Lastly, we show the incremental compilation process of Modula-2 code into GUIDE objects. This information is necessary to understand how the debugger was implemented.

3.1 Smalltalk

3.1.1 The Advantage

Smalltalk is an attractive programming environment for prototyping research. It has been used successfully in prototyping an animation system which visualizes programs and algorithms [London 85]. The designers of the system conclude, "Since Smalltalk is an open system designed with much attention to the human interface and with many elegant facilities at our disposal (modifiable, if necessary), it is not surprising that we have been able to use it successfully in our exploratory and prototyping research." The user interface and the graphics provided by Smalltalk are important to GUIDE's effective use of a graphical users interface [Szafron 86b]. The user interface implementation of Smalltalk, which is called the Model-View-Controller (MVC) system, allows the display representation of an object to be maintained in a view. The view monitors the state of the object and reflects the changes in its own window. If the view dynamically reflects the changing state of the object during the execution of a program, an animation effect is

achieved. This aspect of the user interface is an invaluable tool to the dynamic display of data structures in the GUIDE debugger. Details of the MVC system will be discussed subsequently in this chapter.

Besides the rich graphics library and the user interface, Smalltalk provides a facility to monitor changes made to the system. All changes are recorded automatically and can be migrated to another Smalltalk environment. It is therefore possible to have the GUIDE prototype developed by a group of designers at the same time, each working on a different part of the prototype and each keeps his own history of changes. Later, they can merge their changes together and thus achieve synchronization of a design stage.

Even though Smalltalk, being an interpretive language, is often criticized for its speed, our Smalltalk-80 interpreter runs suitably fast on a Sun-workstation and especially on a Sun-3. Further, the versatility and flexibility of Smalltalk far outweighs its speed deficiency in the development of the GUIDE prototype.

3.1.2 The Language

Smalltalk is an object-oriented language. In the world of object-oriented programming, everything is represented by objects. An object is a self-contained entity which consists of two parts: its state and its behavior. The state of an object is described by its own private memory, using instance variables. The behavior is a set of operations to manipulate that data. In terms of a conventional language, an object contains the data structures as well as the procedures to manipulate those data structures.

The executable instructions to perform the operations are called *methods*. A method is invoked by sending a message to an object which is called the message *receiver*. A message consists of the receiver, the message's name (called the *selector*) and perhaps a set of arguments. The message specifies which operation is desired, but not how that

operation should be performed. The receiver of the message determines how to perform the requested operation by executing the corresponding method.

The only way to manipulate an object's private memory is by sending messages to the object. The object then responds to the message and carries out the operations on its private memory. The internal structure of an object is completely hidden from the rest of the system. Thus the message passing mechanism ensures the modularity of the system.

Let's consider an example,

```
anInteger ← 2 + 3.
```

The addition operation is performed by sending a message to the object representing the number 2. The message specifies that the desired operation is addition and also specifies that the object representing the number 3 should be added to the receiver. The message does not specify how to perform the addition. The receiver will carry out the operation by invoking the method for addition. The method, after completing the computation, returns the object representing the number 5 and the system binds the variable, anInteger, to the object 5.

Every object in the system is an instance of a *class*. A class describes the properties of a collection of similar objects. These objects with similar properties are called *instances* of the class. The class description includes a specification of its instances' private memories, called *instance variables*, and the set of methods to manipulate those instance variables. The instances of a class all have the same number of instance variables and they all respond to the same set of messages. It is the value of the instance variables that makes an instance distinct from another instance of the same class.

For example, class Rectangle defines two instance variables, origin and corner. The two instances of Rectangle shown in Figure 3.1 store different values in the instance variables and therefore have different sizes and locations. Both of them understand the same set of messages such as move, origin, area, etc.

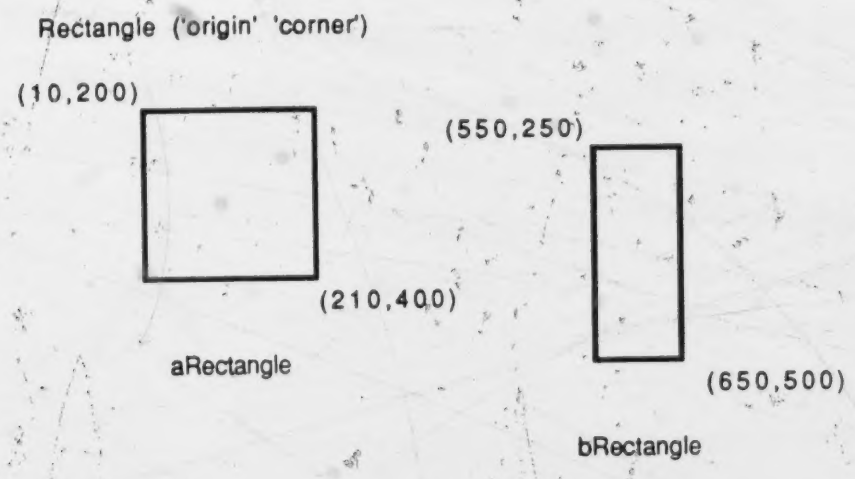


Figure 3.1 Instances of class Rectangle.

Classes are structured in a tree-like hierarchy. Except for the class Object, which is at the top level of the hierarchy, every class is a *subclass* of one or more *superclasses*. Smalltalk restricts a subclass to only one superclass. A subclass inherits all the superclass' properties, i.e. the methods and instance variables. However, the subclass can add more methods and instance variables to its description. It can also modify its behavior by redefining a method with the same name, effectively overriding the superclass' method. Thus, a subclass is a specialization of its superclass. A superclass is usually a generic description of several subclasses and each subclass refines the properties of the superclass.

This inheritance property of object-oriented languages is a powerful tool in system development. A large portion of the code is reusable, reducing both the code size and development time.

3.2 User Interface

Smalltalk-80 has a user interface architecture known as the model-view-controller (MVC) system. For example, pop-up menus, text windows, code editors, and process scheduler are described by the MVC paradigm. GUIDE follows this scheme to construct its user interface. The MVC structure has three components:

- model — the object that is being viewed and manipulated,
- view — the external display of the model to the user, and
- controller — an object that captures and interprets user input to the model.

A model may be any object in the system. A view is opened on the model by creating instances of subclasses of View and Controller and connecting them to one another as shown in Figure 3.2.

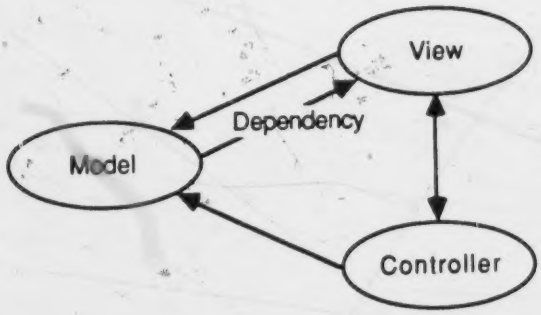


Figure 3.2 The Model-View-Controller structure.

The view handles all windowing manipulations such as displaying, framing, labeling, scrolling, bordering, and mapping local view coordinates to screen coordinates. It displays a graphical representation of the model in the window. The controller accepts the user input from the mouse or the keyboard and handles window scheduling; the set of all controllers is polled by the process scheduler to determine which window wants control.

In this structure, the model has no explicit knowledge of its view or controller. Rather, it communicates with the view via a dependent list. When a view is opened on a model, the view is added to the dependent list kept by the model. When a view is closed, it is removed from the list. There may be more than one view opened on the same model, in which case all the views are on the model's dependent list. The model typically does not know how many or who these dependents are. When the model changes its private state, an update message is broadcast to all objects in the dependent list. Every dependent is then aware of the change in the model and acts accordingly. Usually an argument that specifies the model's changed aspect is sent along with the broadcast message. Only those dependents who are interested in that specific aspect will act upon the change. Other dependents simply ignore it.

If a model is removed from the system, all of its dependents are also informed. All views that are opened on the model will be closed and subsequently their windows removed from the screen. The MVC paradigm achieves the synchronization in a well-defined manner.

3.3 GUIDE Architecture

3.3.1 GUIDE User Interface

All of GUIDE's internal structures such as header text, statement tree, and symbol table are manipulated using the MVC paradigm. However, the model is not the internal structure

itself but a browser on the internal structure. The browser is a link between the internal structure and the user interface. The browser, view and controller coexist together and more than one browser can be opened on the same internal structure. With the same model-view dependency concept, all browsers on the internal structure are on the dependent list of the internal structure. Figure 3.3 depicts the relationship between the internal structure and the MVC structure.

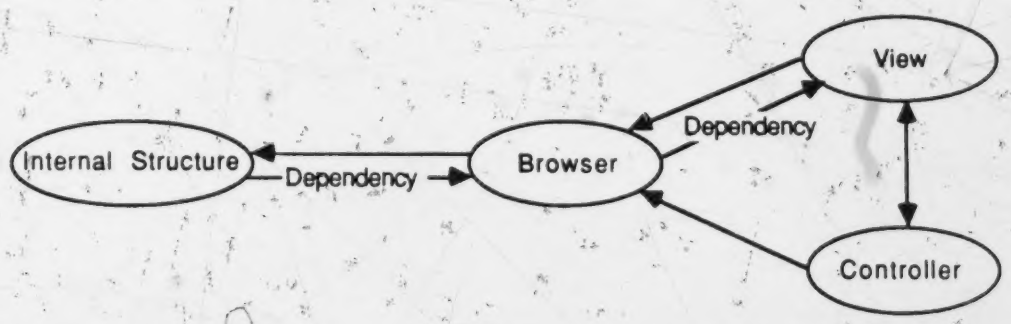


Figure 3.3 The relationship between the internal structure and the MVC structure.

When the private state of the internal structure changes, each browser will receive the update message broadcast by the internal structure; and in turn, each view associated with the browser will be updated and the changes will be reflected in the windows of the views.

3.3.2 Structure Browsers

Each internal structure has its own browser class. For example, the statement tree's browser class is GuideCodeBrowser and the symbol table's browser class is DeclarationBrowser. All of GUIDE's browser classes for internal structures are subclasses of StructureBrowser. Each browser class provides the pop-up menus and methods for manipulating its corresponding internal structure. The pop-up menus are context sensitive. Depending on the current location of the insertion point or the current selected structure, the browser generates a menu consisting of only allowable commands. It is not possible for the user to illegally edit the internal structure. An instance variable

called currentStructure in the browser keeps track of the current insertion point or the current structure, and is able to provide the browser with all the information to generate the context-sensitive menu. With the effective use of pop-up menus, syntax directed editing is achieved. In the following sections, we will describe some of the subclasses of GuideStructure that are of particular importance.

3.3.3 Guide Structures

Any internal structure in GUIDE that can be manipulated using the syntax directed editor must be an instance of a subclass of GuideStructure. Examples of the subclasses are GuideParseNode and GuideSymbol. Each structure is able to display itself textually and define its boundary in the program text. The boundary is an integer pair that marks the starting index and stopping index in the program text. It is used for structure selection and highlighting during syntax directed editing. To select a structure, the user clicks on that structure in the program text. The controller translates the position of the mouse cursor into an index of the program text. The structure whose boundary encloses the index is selected and highlighted.

3.3.3.1 StructureList

A special subclass of GuideStructure is the StructureList which is capable of holding multiple GuideStructure(s). For example, the symbol table of a procedure is a StructureList which holds the local declarations of the procedure such as types and variables. Another example is the body of the IF statement.

3.3.3.2 GuideParseNode

GuideParseNode is the superclass of all parse node classes constructed for the Modula-2 language. Every node in the statement tree is an instance of a subclass of GuideParseNode.

The nodes are arranged in a way that represents the logical flow of the code. In addition to the editing capabilities inherited from GuideStructure, a GuideParseNode can evaluate itself within its context. Each Modula-2 statement has a corresponding subclass under GuideParseNode because each has a different runtime behavior. Details of the code interpreter will be discussed in Chapter 4.

GuideParseNodes are categorized into five major subclasses based on their functionality. Figure 3.4 shows the class hierarchy of GuideParseNode. The notation we used to represent classes and their hierarchical relationship is borrowed from Smalltalk-80. A class name is an identifier starting with an uppercase letter. Following the class name and enclosed in brackets is a list of instance variables defined in that class. Under the class name and indented one tab position are the subclasses of that class.

3.3.3.2.1 GuideExpressionNode

A GuideExpressionNode returns a literal as the result of its evaluation. Examples are unary expressions, binary expressions, array element accesses, and variable accesses. A GuideExpressionNode cannot exist on its own; it must be a substructure of other parse nodes such as the right hand side of an assignment statement or the condition expression of an IF statement.

3.3.3.2.2 GuideEditableNode

Parse nodes of subclasses of GuideEditableNode will require additional information from the user to complete their specification. When the user inserts such a node into the statement tree, a template is first inserted into the code and then a dialog box is opened on the structure (see Section 2.2.1.1) to prompt the user for more information. Assignment statements and procedure calls are examples of GuideEditableNodes.

```
GuideParseNode ()
    BreakPoint ()
    GuideCompoundNode ('body' )
        GuideBlockNode ()
        GuideControlNode ('expression' )
            GuideClauseControlNode ('clause' )
                GuideCaseControlNode ()
                    GuideCaseClauseNode ()
                    GuideCaseNode ()
                GuideIfControlNode ()
                    GuideElseIfNode ()
                    GuideIfNode ()
            GuideForNode ()
            GuideRepeatNode ()
            GuideWhileNode ()
            GuideWithNode ()
        GuideElseNode ()
        GuideLoopNode ()
    GuideEditableNode ('isTemplate' )
        GuideAssignmentNode ('leftHandSide' 'rightHandSide' )
        GuideCaseLabelListNode ('caseLabelList' )
        GuideEditableExpressionNode ('expression' )
            GuideCaseExpressionNode ()
            GuideConditionNode ()
            GuideRecordExpressionNode ()
        GuideForControlNode ('variable' 'initialValue' 'finalValue' 'step' 'cachedFinalValue' )
        GuideProcedureCallNode ('argumentList' 'procedure' )
        GuideReturnNode ('returnValue' )
        GuideReturnTypeNode ('returnType' )
    GuideExitNode ()
    GuideExpressionNode ()
        GuideArrayAccessNode ('array' 'indexes' )
        GuideBinaryExpressionNode ('leftOperand' 'operator' 'rightOperand' )
        GuideLiteralNode ('literal' )
        GuidePointerAccessNode ('pointer' )
        GuideRecordAccessNode ('record' 'field' )
        GuideSetAccessNode ('setType' 'elements' )
        GuideUnaryExpressionNode ('operator' 'operand' )
```

Figure 3.4 Class hierarchy of GuideParseNode.

3.3.3.2.3 GuideCompoundNode

An instance of GuideCompoundNode has a body which is a StructureList that holds a sequence of statements. The body of a compound statement may be empty. Examples are **IF** statements, **WHILE** statements, **LOOP** statements, **REPEAT** statements, and **ELSE**

clauses of IF statements. StructureList provides the functions to insert, remove and select statements in the body of these nodes.

3.3.3.2.4 GuideExitNode

GuideExitNode represents the **EXIT** statement. The **EXIT** statement exists only within the body of a repetitive statement such as **LOOP** or **WHILE**. There is no subclass of GuideExitNode.

3.3.3.2.5 BreakPoint

A BreakPoint is not a Modula-2 statement. It is used during debugging to mark the location where execution should be suspended. When execution is suspended, a user may inspect or modify the state of the executing program, such as the contents of the variables. A user may also proceed to step through the code slowly to monitor the effects of execution.

3.3.3.3 GuideSymbol

GuideSymbol represents all the symbols used by the Modula-2 language such as constants, variables, types, procedures, and modules. Each symbol is identified by a name. The scope of a symbol is the procedure or module within which it is declared. Symbols with the same name can exist in different scope. There are three major subclasses of GuideSymbol — GuideDeclarationSymbol, GuideEnvironmentSymbol and GuideImportSymbol. Figure 3.5 summarizes the class hierarchy of GuideSymbol.

```
GuideSymbol ('name' 'isTemplate' )
    GuideDeclarationSymbol ('isPublic' )
        GuideTypedDeclarationSymbol ('type' )
            GuideArrayTypeSymbol ('indexType' )
                GuideStringTypeSymbol ()
            GuideEquivalenceTypeSymbol ()
            GuideOpenArrayTypeSymbol ()
            GuideParameterSymbol ('mode' )
            GuidePointerTypeSymbol ()
            GuideRecordFieldSymbol ()
            GuideSetTypeSymbol ()
            GuideSubrangeTypeSymbol ('lowerBound' 'upperBound' )
            GuideValuedDeclarationSymbol ('value' )
                GuideConstantSymbol ()
            GuideVariableSymbol ()
        GuideUntypedDeclarationSymbol ()
            GuideBaseTypeSymbol ()
                GuideBooleanTypeSymbol ()
                GuideCardinalTypeSymbol ()
                GuideCharTypeSymbol ()
                GuideIntegerTypeSymbol ()
                GuideLiteralNumberTypeSymbol ()
                GuideRealTypeSymbol ()
                GuideTypeTypeSymbol ()
                GuideUnassignedTypeSymbol ()
            GuideProcedureTypeSymbol ('parameterTypes' 'returnType' )
            GuideRecordPlaceHolder ('identifier' 'fieldList' )
            GuideRecordTypeSymbol ('fieldList' )
            GuideScalarTypeSymbol ('valueList' )
            GuideScalarValueSymbol ('ordinalValue' )
    GuideEnvironmentSymbol ('symbolTable' 'environmentList' )
        GuideContextSymbol ('comment' 'codeBody' 'declarationList' 'activeContext' )
            GuideLanguageSymbol ()
            GuideModuleSymbol ('importList' 'exportList' )
            GuideProcedureSymbol ('parameterList' 'returnType' 'isPublic' )
        GuideProjectSymbol ()
        GuideSystemSymbol ()
```

Figure 3.5 Class hierarchy of GuideSymbol.

3.3.3.3.1 GuideDeclarationSymbol

Instances of GuideDeclarationSymbol are identifiers used in standard declarations such as variables, types and constants. Subclasses of GuideDeclarationSymbol are

GuideTypedDeclarationSymbol and GuideUntypedDeclarationSymbol. Typed symbols are those that require a base type, e.g. array types, set types, subrange types, constants, and variables. Untyped symbols do not have a base type. They include the predefined types, i.e. INTEGER, CHAR, BOOLEAN, CARDINAL and REAL, and untyped declarations such as record types and scalar types.

3.3.3.3.2 GuideEnvironmentSymbol

An environment (GuideEnvironmentSymbol) in GUIDE refers to a language (GuideLanguageSymbol), a module (GuideModuleSymbol) or a procedure (GuideProcedureSymbol). This class contains a symbol table which is a StructureList of instances of GuideDeclarationSymbol. The symbol table contains the local declarations of the environment. Currently, the only instance of GuideLanguageSymbol is Modula. Its symbol table contains all the predefined types of the language.

The class GuideModuleSymbol has two instance variables, importList and exportList, which are used for exchanging information with the other modules in the system. The importList is a StructureList of instances of GuideImportSymbol (to be discussed next) and the exportList is a StructureList of symbols that can be seen outside of the module.

3.3.3.3.3 GuideImportSymbol

Import symbols are identifiers that are imported from another module. They are declared elsewhere but must be imported before they can be used locally. GuideImportSymbol acts as a placeholder since it contains a pointer to the imported symbol and a pointer to the symbol's module; it does not have the actual definition of the imported symbol.

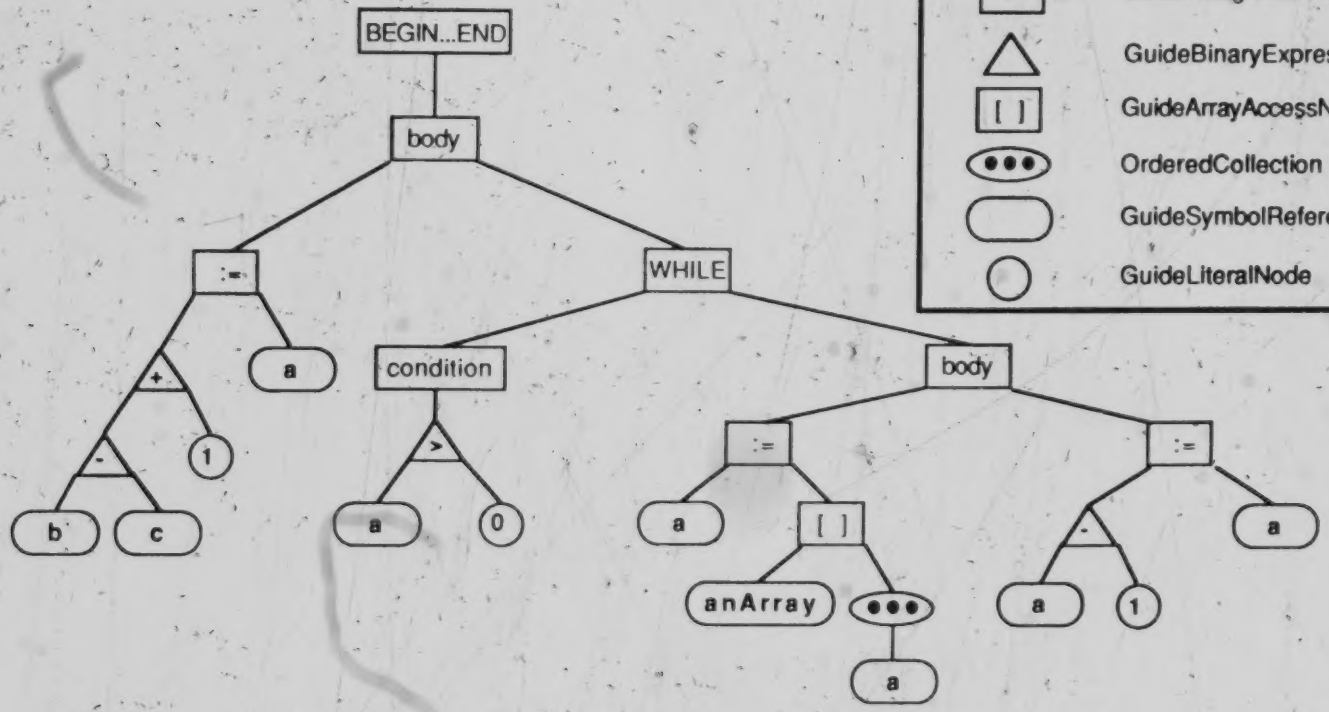
3.3.3.4 GuideSymbolReference

The editing of primitive structure requires the GuideParser to parse the character stream entered by the user in a dialog window (see Section 2.3.1.1). The parser is invoked by the dialog interface when the user chooses accept from the dialog menu. The ability of the parser is limited to parsing expressions and expression lists. Identifiers in the expression being parsed are not bound to any environment yet; the parser is responsible for syntactic error detection only. These unresolved symbols are represented by instances of the class GuideSymbolReference. If a syntactic error is found during parsing, an error message is inserted into the text in the dialog window. If the parsing is successful, the dialog then attempts to bind the unresolved symbols to the current environment. During this symbol binding phase, the semantic validity is checked. When a semantic error is found, an alert window is generated (Section 2.3.2). If the user input is free of semantic errors, then GuideSymbolReference is bound to the actual symbol by setting a pointer to it and the primitive template in the program text is replaced by the resolved primitive structure.

3.4 Incremental Compilation

During syntax-directed editing of the program code, instances of GuideParseNode are inserted into or removed from the statement tree. As the user is coding the program, the statement tree is gradually built. Each parse node in the statement tree is able to execute itself, i.e. each parse node is an executable instruction used internally by GUIDE. The statement tree is in fact the compiled version of the program code. What the user sees in the code view is a textual representation of the statement tree. Figure 3.6 shows an example of a statement tree.

```
BEGIN
 a := b - c + 1
 WHILE a > 0 DO
  anArray[a]  :=  a
  a  :=  a  -  1
 END WHILE
END
```



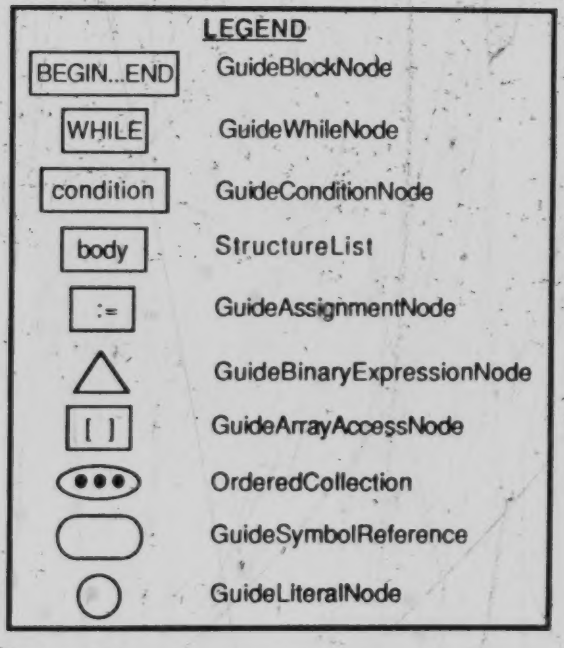


Figure 3.6 Example of a statement tree.

3.5 Summary

In this chapter, we have presented an overview of object-oriented programming in Smalltalk. The properties of class hierarchy and inheritance of object-oriented languages promote code re-usability and modifiability. The design of the GUIDE architecture takes advantage of these properties. Abstract classes are created so that subclasses can specialize their superclass' behavior. We have described the class hierarchy structure of the parse nodes and the symbols.

The user interface structure, called the model-view-controller (MVC) paradigm, used in Smalltalk provides a modular architecture of window display, user input handling and object manipulation. GUIDE uses this paradigm to produce a consistent user interface throughout its implementation. We have also explained how syntax directed editing is achieved and how synchronization of displays is done using the MVC paradigm. Incremental compilation is done when the user is editing the code in the code view during which the statement tree is built.

CHAPTER 4

RUNTIME ENVIRONMENT IN GUIDE

This chapter describes the runtime environment in GUIDE. We will present the implementation of activation record which contains all the information about the internal state of an executing program. We will discuss how an activation record is initialized and how memory is allocated when a procedure is invoked. Finally, we will show how execution is performed in an object-oriented manner. Before moving on to the details, it is necessary to review the runtime behavior of Modula-2.

4.1 The Runtime Behavior of Modula-2

Modula-2 is a block structured language that is ALGOL-like. ALGOL-like languages have common properties in their runtime behavior. Most of the discussions that follow are applicable to other ALGOL-like languages as well.

4.1.1 Activation Record

Modula-2 allows a program to be composed of a number of procedures. Variables declared within a procedure are local to that procedure and global only to procedures that are nested within itself. A procedure is activated when it is invoked by a procedure call from another procedure. The activation of a procedure is composed of two parts: a code segment and an activation record. The code segment consists of the static executable instructions of the procedure. The activation record contains all the information required to execute the procedure plus the storage allocated for the local variables declared in that procedure.

Modula-2 also supports recursion. Hence, several activations of the same procedure may exist at the same time. All the activations have the same code segment but different activation records. This implies that an activation record must be allocated dynamically when a procedure is activated.

When a procedure completes its execution, control is returned to the calling procedure. Its activation record is no longer needed. The space occupied by the activation record can be released and made available for new activation records. Since the activation record that is deallocated is also most recently allocated, activation records are usually allocated on a last-in-first-out manner, that is, using a stack-based storage.

4.1.2 Dynamic Link

In order to return control to the calling procedure, activation records must contain the information to address the next instruction to be executed upon return and to access the caller's activation record which will become active. The pointer to access the caller's activation record is called the dynamic link. The chain of dynamic links from the currently active activation record is called the dynamic chain. The dynamic chain represents the calling sequence of procedures.

Figure 4.1a shows the layout of a program. In the figure, the following chain of calls is issued:

Procedure A calls procedure E.

Procedure E calls procedure F.

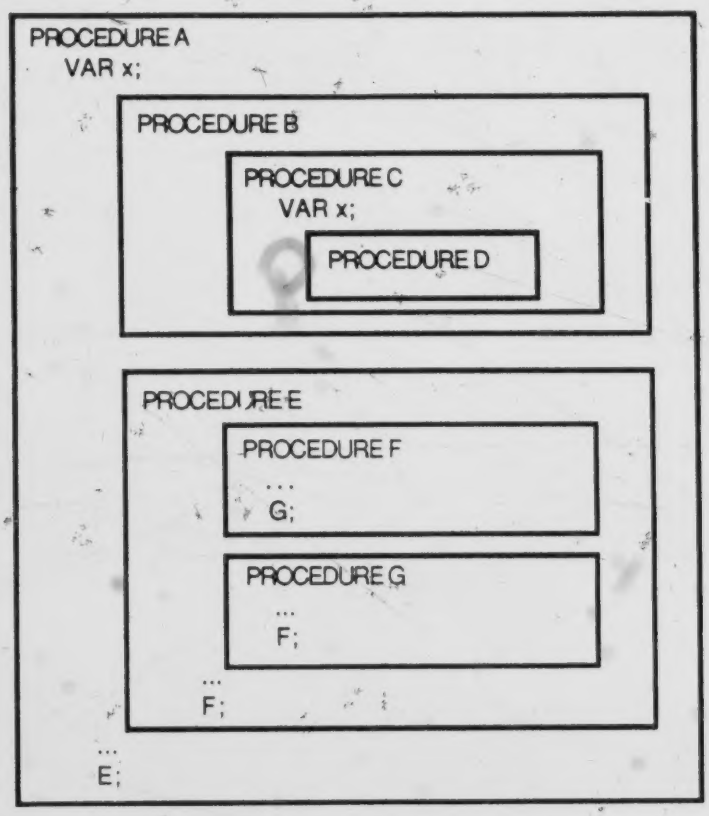
Procedure F calls procedure G.

Procedure G calls procedure F.

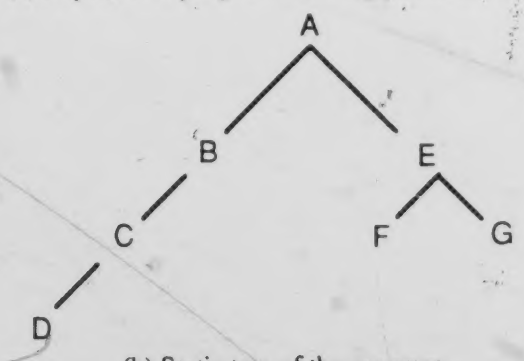
Procedure F calls procedure G.

Procedure G calls procedure F.

Note that procedures F and G are mutually recursive. The static structure of the program is shown in Figure 4.1b. Figure 4.1c shows the dynamic links and chain on the execution stack.

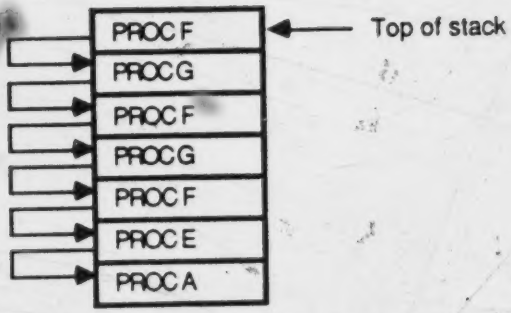


(a) Layout of a program showing procedure calls



(b) Static tree of the program

Figure 4.1 Example of procedure calls.



(c) Execution stack showing dynamic links

Figure 4.1 Example of procedure calls. (cont.)

4.1.3 Static Link

The static scoping rules specify that a variable declared within a procedure is local to the procedure and global to the procedures nested within it. In Figure 4.1a, procedures B and E are statically enclosed within procedure A; procedures F and G are statically enclosed within E; C within B; and D within C. If a local variable is declared in a procedure P, it is visible in P, but not in the procedures that enclose P. However, it is visible (with one exception) to all the procedures that are statically nested within P.

The static tree can be used to determine the visibility of variables to any procedure; the local variables of a node in the tree are visible only to itself and its descendents. For example in Figure 4.1b, a local variable in procedure B is not visible to A, E, F and G; it is visible to B, C and D. It is local to B and global to C and D. The exception to the above rule occurs when a variable local to a procedure is given the same name as a variable declared in an enclosing procedure. In such a case, the rule is that the local declaration overrides the global declaration. For example, referring to Figure 4.1a, variable x is declared both in A and C. Any references to x within A, B, E, F and G refer to the variable x declared in A, whereas any references to x within C and D refer to the variable x declared in C. We can

also determine the static scope of any given procedure from the static tree; a node on the tree can access the local variables (with the above exception) of its ancestors. For example, procedure F can access the local variables of A and E.

When a procedure is activated, its local variables are implicitly created in the activation record. In referencing a local variable of a procedure, it is sufficient to examine the activation record of that procedure since the storage of all local variables are allocated with the activation record. However, a reference to a global variable requires access to the activation record on the call stack that contains that global variable. This access is made possible by having each activation record contains a pointer (static link) down the stack to the activation record of the statically enclosing unit in the program. In other words, the static link represents the back pointer of a node to its parent in the static tree of Figure 4.1b. Figure 4.2 shows the static links for the example of Figure 4.1.

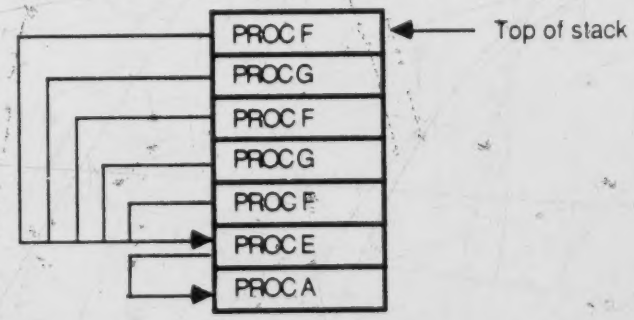


Figure 4.2 Static links for Figure 4.1.

Referencing global variables can be done by searching down the static links until a binding is found. In the example, a reference to x within D is bound to a storage location within C's activation record and a reference to x within E is bound to a location within A's activation record. Note that static links are not the only mechanism for accessing non-local variables. For example, displays can also be used [Ghezzi 82].

So far we have assumed that a procedure is accessible only from the procedure that encloses it and therefore the above scheme would work. But this is not the case in Modula-2. In Modula-2, a module allows other modules to access its procedures if they are exported. If a procedure is called from outside of its own module, it will not be able to find the activation record of its module on the execution stack. It is because the module body is not activated when the call to that procedure is made. The global variables declared in the module body become inaccessible to the procedure.

To solve this problem, the module body must somehow be executed first and its activation record not be deallocated after the execution is completed. This is done when the main program is invoked; all of the imported modules' bodies are executed in the same order they are imported. A module is initialized only once even if it is imported by many modules. The activation records of the imported modules may be kept on the same execution stack or on another global stack. They continue to exist until the execution of the program is completed. When an imported procedure is called, its activation record must set up the static link to the activation record of its module.

The management of activation records discussed above is an implementation choice and is not strictly implied by the semantics of Modula-2. Variations do exist and some are more efficient; but the implementation of the debugger view follows closely to the above description.

4.2. Activation Record

To simulate the runtime behavior of Modula-2, we will need to have activation records and an interpreter to carry out the execution. The class ActivationRecord contains all the information about the internal state of an executing program. It has several instance variables that can be found in a conventional activation record. In addition, it also has

instance variables for runtime execution (Section 4.7) and for keeping an execution history (Section 6.3). The following instance variables are defined in ActivationRecord:

caller	the caller's activation record, i.e. the dynamic link.
scope	the activation record of the enclosing unit, i.e. the static link.
procedure	the context being executed.
valueTable	a dictionary which binds local variables to their storage.
nextStatement	the next statement to be executed.
importList	pointer to a global stack of activation records of imported modules.
stack	a stack to hold the results of evaluation.
history	an object for keeping the execution history.

The following sections will discuss how these instance variables are initialized.

4.3 Context Switching

A context switch is necessary when a procedure is invoked or when a procedure is returning control to its caller. The current activation record is responsible for initiating the context switch. When a procedure call is made, the arguments are first evaluated and their values are pushed on the evaluation stack. The following message is then sent by the current activation record to create a new one:

```
newActivationRecord ← ActivationRecord new
        caller: self
        procedure: nextStatement procedure symbol
        importList: importList.
```

In Smalltalk, self refers to the sender of the message. In this case, it is the current activation record which is the caller of the new activation record. This message is sent when the nextStatement to be executed is a procedure call (a GuideProcedureCallNode). The message nextStatement procedure symbol returns the context (a GuideModuleSymbol or GuideProcedureSymbol) to be invoked. The argument importList is the instance

variable that points to the global stack for storing the activation records of imported modules.

After the new activation record is created, its instance variables caller, procedure and importList are initialized with the values of the arguments. It then sets up the static link and allocates storage for the local variables (including the formal parameters) of the new context.

There are three kinds of parameters in Modula-2: IN, OUT and IN/OUT. The IN and IN/OUT parameters are initialized by copying the values of the arguments from the caller's evaluation stack. Finally, the first statement to be executed is initialized and the new activation record takes over the execution control. Upon completion of the procedure call, the values of the OUT and IN/OUT parameters are copied back to the arguments. Control is then returned to the caller.

4.4 Dynamic Link

An activation record is usually invoked by another activation record, i.e. the caller. The first activation record is however invoked by the debugger, in which case the caller instance variable is undefined. On this first invocation, the module of the procedure must be initialized first and then kept in the instance variable importList. If the module has imported other modules, each of the imported modules are initialized in turn and their activation records are added to the importList. The importList of all activation records points to the same global stack and therefore the activation records of initialized modules are all kept in the same place. To ensure that each module is initialized only once, the importList is searched to make sure that an activation record of that module does not already exist.

4.5 Static Link

The instance variable scope is the static link which points to the activation record of the statically enclosing unit of procedure. During initialization of an activation record, it is initialized by searching down the dynamic chain. If the scope is not on the dynamic chain, then it must be on the global stack pointed to by the importList.

4.6 Storage Allocation

The instance variable procedure is used to access the context's symbol table. From the symbol table, storage for the local variables is allocated and stored in valueTable. valueTable is a Dictionary that associates the variables with their allocated memory storage.

A storage is represented by an instance of the class GuideStorage. A GuideStorage provides the storage space, keeps a record of its modification history and displays itself graphically. Because each data structure type may allocate memory and display itself differently, subclasses of GuideStorage are created for each type. Figure 4.3 shows the class hierarchy of GuideStorage. We will postpone the discussion of data structure display and history recording until Chapter 6 since they are related to debugging.

```
GuideStorage ('variable' 'type' 'display' 'activationRecord' )
    GuideCompositeStorage ('boundingBox' 'elements' 'elementBoxes' )
        GuideArrayStorage ()
        GuideRecordStorage ()
        GuideSetStorage ('history' 'historyIndex' )
    GuideLiteralStorage ('value' 'history' 'historyIndex' )
        *GuidePointerStorage ()
        GuideStringStorage ()

*GuidePointerStorage is not implemented yet.
```

Figure 4.3 Class hierarchy of GuideStorage.

GuideLiteralStorage allocates space for all basic types (CARDINAL, INTEGER, BOOLEAN, REAL and CHAR) and scalar type. These types are primitive structures that do not consist of any sub-structures. String is a special case that, by definition, it is an array of characters; however, it is often treated as a primitive type. Very seldom do programmers need to inspect individual characters of a string. Implementing it as a subclass of GuideLiteralStorage does not prevent one from accessing and manipulating individual characters, but when the string is viewed by an inspector window, its elements will not be listed as in the case of an array (Figure 5.6a). The advantage is speed gain and less memory usage. Otherwise instances of GuideLiteralStorage would have to be allocated for the characters which will require more time and memory.

GuideCompositeStorage is for the type of composite data structures that encompass other data structure types. They are ARRAY, RECORD and SET. The base type of SET must be a basic type except REAL. Individual elements of a set are not accessible but they are displayed as shown in Figure 5.7. Because its elements are displayable, each one must be allocated as an instance of GuideLiteralStorage.

ARRAY and RECORD are more complex because their base type can be another composite type. Common examples are 2-dimensional arrays and arrays of records. The storage is allocated recursively. Storage space is first allocated to hold the elements' storage, then each element is in turn requested to allocate space for itself. For example, an array of 10 records would be allocated using a GuideArrayStorage that has an array of 10 instances of GuideRecordStorage; each instance in turn allocates space for itself. The allocation process is stopped when finally a literal type is encountered and an instance of GuideLiteralStorage is allocated.

Subrange types and equivalence types use the storage class of their base type for allocation. For example an equivalence type of a record type uses the class GuideRecordStorage.

Currently, because POINTER type is not supported yet, GuidePointerStorage is also not implemented. However, the allocation scheme would be quite straight forward: a space is allocated to store the pointer to another storage. The storage that it is pointing to must already be allocated by some other means, such as a call to NEW. It can be implemented as a subclass of GuideLiteralStorage so that it would inherit most of the memory management methods while overriding the display method so that it is displayed as a box with an arrow pointing to another storage (Section 6.1.3).

4.7 Code Interpretation

Because an activation record contains all the information necessary for execution, it is appropriate for the activation record to control its flow of execution. GUIDE does not have a centralized interpreter as in conventional debuggers. Instead, each activation record is responsible for controlling its own execution. An activation record can execute its code, restart itself, step forward, step backward and replay its execution history. We will discuss the implementation of the primitive execution routine in this section and leave the discussion on the debugging-related functions until Chapter 6.

Because the statement tree is already in compiled form, there is no need for the activation record to decode the statements in the program code. The actual execution is carried out by the nodes in the statement tree. Each parse node knows what to do when it receives the message to execute itself.

Before execution can begin, the activation record must search for the first statement to be executed. This is done by sending the message firstStatement to the nodes in the tree.

For example, referring to Figure 3.6, the GuideBlockNode at the top of the tree first receive the message. It in turn sends firstStatement to its body, a StructureList. The body passes the message to the first node in its list, which is the GuideAssignmentNode. Eventually, the parse node at the bottom of the tree, which is the GuideSymbolReference representing the variable b, receives the message and responses to it by returning itself. The activation record saves this parse node in the instance variable nextStatement and execution may start.

Figure 4.4 shows the flow chart of the primitive execution routine implemented by ActivationRecord. The activation record first checks whether a context switch is necessary. Then it keeps evaluating nextStatement and obtaining the statement to be evaluated next and saves it in nextStatement. This execution loop terminates when an error occurs or when nextStatement is not a primitive statement any more. A primitive statement is one that can be evaluated immediately without any operation, i.e. variables, constants and literals. Control is then returned to the debugger which is the sender of this routine. Depending on which stepping forward command is specified by the user, the debugger may choose to continue the execution or it may suspend itself so that the user can specify further actions.

The following message is sent to nextStatement to evaluate itself within the current activation record:

```
aValue ← nextStatement valueIn: self.
```

aValue saves the result of the evaluation so that it can be used to determine the flow of execution. During the evaluation, temporary results are pushed on the evaluation stack (instance variable stack) of the current activation record. As an example, the method implemented by GuideBinaryExpressionNode is shown here.

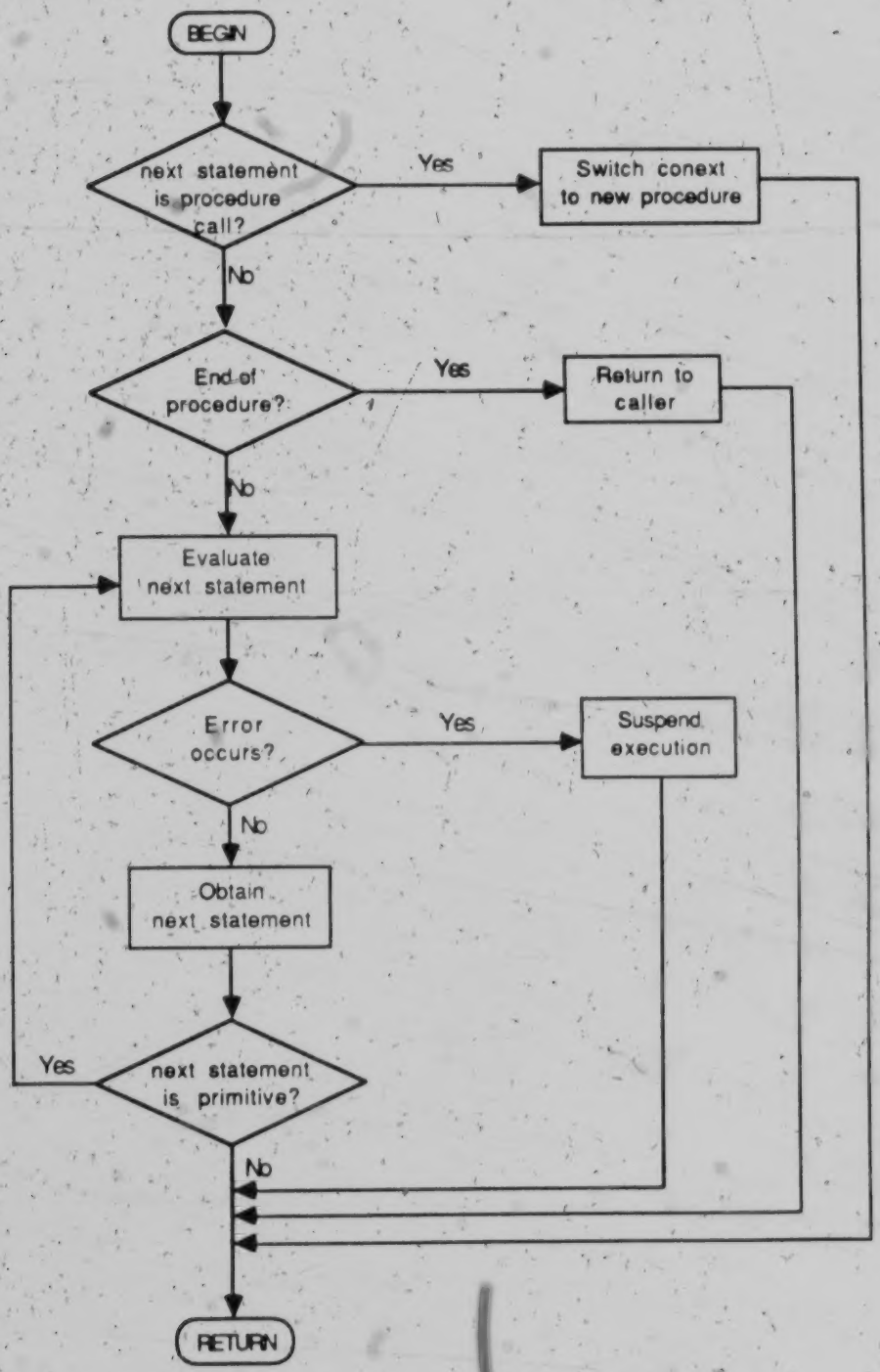


Figure 4.4 Flow chart of primitive execution routine.

GuideBinaryExpressionNode

```
valueIn: anActivationRecord
    "Evaluate myself within the context of anActivationRecord."

    | leftResult rightResult result literalNode |
    rightResult ← anActivationRecord pop literal.
    leftResult ← anActivationRecord pop literal.
    result ← self evaluateLeft: leftResult right: rightResult.
    literalNode ← GuideLiteralNode new literal: result.
    literalNode scope: self.
    anActivationRecord push: literalNode.
    ↑result
```

When this message is sent to GuideBinaryExpressionNode, the operands are already evaluated and their results are pushed on the evaluation stack of anActivationRecord. GuideBinaryExpressionNode removes the results from the stack and performs the binary operation on them. A new instance of GuideLiteralNode is created to store the result of the operation and then it is pushed on the evaluation stack.

If there is no runtime error, then the statement to be evaluated next is obtained:

```
nextStatement ← nextStatement scope
        statementAfter: nextStatement
        value: aValue.
```

This message is sent to the scope of nextStatement because only it will know which statement comes after the current one. The result of the previous evaluation, aValue, is passed along with the message so that the scope can determine the flow of execution. Perhaps it is easier to understand by looking at the method implemented by GuideWhileNode:

GuideWhileNode

```
statementAfter: aStatement value: aValue
    "Answer the next statement to be executed after aStatement."

    aStatement == expression
        ifTrue:
            [aValue
                ifTrue: [↑body firstStatement]
```

```
                    ifFalse: [↑scope statementAfter: self value: nil]]
        ifFalse:
            [aValue == nil
                    ifTrue: [↑expression firstStatement]
                    ifFalse: [↑scope statementAfter: self value: aValue]].
```

GuideWhileNode first tests whether the previous evaluated statement is the condition expression. If it is, then aValue contains the result of the condition. If the condition is true, then the first statement of the while body is returned to be evaluated next; otherwise, the while loop is exited and its scope is requested to return the next statement after the while loop. If the previous statement is not the condition expression, then it must be the while body. The body may have a non-nil result if the while loop is to be exited immediately, such as the encounter of a RETURN statement. In which case, the scope is notified with the non-nil result. If the result is nil, then it loops back to test the condition again.

Let's consider one more example:

GuideBinaryExpressionNode

```
statementAfter: aStatement value: aValue
    "Answer the next statement to be executed after aStatement."

    aStatement == leftOperand
            ifTrue: [↑rightOperand firstStatement]
            ifFalse: [↑self]
```

One interesting point about this method is that GuideBinaryExpressionNode returns itself as the next statement to be executed after both operands are evaluated. This is necessary so that it can receive the message valueIn: in order to carry out the operation on the operands.

The methods firstStatement, valueIn: and statementAfter:value: are implemented by the parse node to execute the code and determine the flow of execution. There is no need for another code interpreter to decode the statement tree or to perform any flow control such as branching.

4.8 Summary

The management of activation records is implemented in a conventional way, i.e. dynamic link and static link are set upon the activation of a procedure, and memory storages are allocated to the local variables and stored in the activation record. Instead of having a centralized interpreter, each activation record is responsible for controlling its own execution and managing its own storage. Object-oriented design has distributed the execution to each parse node. Each parse node knows how to execute itself and to return the next statement to be executed. The activation record controls the flow of execution by repeatedly sending messages to each parse node to execute itself and to obtain the next statement.

CHAPTER 5

AN OVERVIEW OF THE GUIDE DEBUGGER

As mentioned in Section 1.4, the GUIDE debugger is a program visualization system which visualizes both code and data dynamically. It allows the user to interactively execute, debug and test a program. Data structures in the program are displayed pictorially and they can be interactively examined and modified. As the program is being executed, highlights move through the code to indicate the flow of execution. The displays of data structures are updated immediately to reflect the actual state of the program.

In this chapter, the motivation for the GUIDE debugger is first presented and the desired features are listed. Finally, the user interface of the GUIDE debugger is described.

5.1 Motivation for a Debugger

The five GUIDE views (header, code, declaration, interface and context) described in Chapter 2, present a graphical interface to manipulate the internal structure of a context. Although the internal structure is guaranteed to be syntactically and semantically correct, the logical correctness of the program is not addressed. However, logical errors are even more difficult to locate than syntactic and semantic errors. It has been estimated that fifty to ninety percent of the programming task is spent in debugging [van Tassel 74, p.117]. An integrated programming environment is not complete without a debugger.

Debugging is a creative activity. It "demands intuitive leaps, conjectures, experimentation, intelligence, and freedom" [Beizer 84]. A debugger is only a diagnostic tool used to control and examine the internal state of an executing program. It cannot substitute any

thinking for the user. Therefore the GUIDE debugger, like any other debugger, cannot guarantee the logical correctness of a program, but with its effective use of graphics and many useful features, it can help to reduce the software development time significantly.

5.2 Desired Features

Myers [Myers 80, pp.5-9] lists many of the desired features of debugging and data display systems. The following summarizes his findings and discusses how the GUIDE debugger responds to each feature.

1. Speed

   The most common complaint about many current systems is that they run too slowly. Many powerful systems, such as DLISP [Teitelman 77] and Model's system [Model 79], are seldom used because the response time is too long. Users can adapt to the most complicated interface but are frustrated by long delays.

   The GUIDE debugger, running under Smalltalk interpreter, also have the same problem. But being a prototype system, speed is not a primary concern in the current implementation.

2. Information at the user's level

   "Information about the behavior of a program should be presented to the user at the conceptual level at which the program was written and in terms of the constraints and operations of the programming languages used" [Model 79, p.55]. For example, if the user requests the display of a character, the debugger should display the character rather than a hexidecimal number. If the information is presented at a higher level, the user can devote more resources to the demanding analytic and creative phases of debugging activity [Model 79, p.55]. Furthermore, the system

can help the user to better understand the information if the context of the information is also presented.

The GUIDE debugger achieves this by presenting data structures at as high a conceptual level as possible. For example, a character string is displayed as such rather than as an array of characters; or a printable character is properly displayed but a non-printable character is displayed as an octal number. The context of the code being viewed and the context of the variable being displayed are unambiguously shown.

3. Use of appropriate level of detail

The amount of information produced should be reduced to what the user has requested. When the user is interested in a particular value, he should not have to search through a large amount of output to find it. If further details are desired, the user should be allowed to request it selectively.

In the GUIDE debugger, when the user wants to see the value of a variable, he first selects the context in which it is defined and then selects the variable. Only the contents of the variable is displayed in the debugger view. In addition, if several variables are of interest, a separate inspector window can be opened for each one so that they can be viewed simultaneously.

4. Analogical display

Information should be converted into abstract pictures, such as bar graphs, icons, arrows, tables, etc., for presentation so that it is easier for people to understand. It is known that "analogical displays more effectively utilize the brain's innate capacities" [Myers 80, p 7]. Also, "people can more easily see differences in a

pictorial representation more readily than small differences in print-outs of variable contents" [Sherman 84]. Programmers often use diagrams to explain their data structures to others rather than using numbers alone. Thus analogical displays convey information in a more natural and understandable manner, making the debugging activity more productive.

All data structures in the GUIDE debugger are displayed pictorially. For example, an array of integers is displayed as one would expect as shown in Figure 5.1. Most debugging systems present information in a textual manner; but with the graphics capability of Smalltalk, the data structures can be updated dynamically during the execution of a program. The mouse provides random access to any part of the data structures so that they can be selected and modified easily. This is particularly useful in modifying data structures that involve pointers such as linked lists and trees.

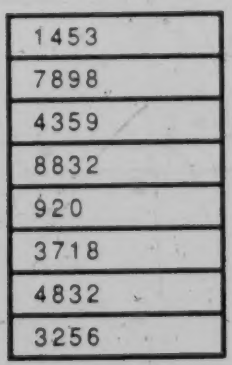


Figure 5.1 Example of a pictorial display of an integer array.

Currently, the debugger does not convert any data structure into a form that is an analogy to the physical world. An analogical display would be application specific and would be impossible to display without any input from the user. For example, the user may want to display an iteration variable in a **FOR** loop as a "percent done

bar" or as a pie chart. The system cannot determine which format the user would prefer. Future version of the debugger will allow the user to define his own display routine (Section 6.1.4) so that this deficiency can be overcome.

5. Automatically generated pictures

Automatically generated pictures can be used for documentation. With manually generated pictures, "there is no guarantee of correspondence with a real computation running on a real machine" [Yarwood 77, p.9]. Thus, the ability to produce pictures automatically is required.

Currently, such a feature is not implemented in the GUIDE debugger. However a utility in Smalltalk has been written to convert a specified area on the screen into Macintosh MacPaint documents which can then be manipulated and reproduced on a printer.

6. Meta-knowledge

A sophisticated system may help the user to understand the information at a more global level. For example, the system might present the value of an array by saying: "All have initial value except for element #17 which equals..." This is much easier to understand than presenting a list of all the values. Even more, the system may be asked to detect if anything unusual had happened during an execution. Such a powerful system, however, would require knowledge and capabilities beyond current available technology.

7. Replay

It would be useful if the history of execution during a debugging session is recorded. A replay of the history at a higher speed can be used to retrace the

execution of a program. Such a facility would be particularly useful if the running time of the program is long. Furthermore, the history can be used to reverse a computation, allowing the user to change something, and then repeat the computation in the new environment. Thus the user can better understand a complex change in data by moving randomly forward and backward around the part of the program which caused the change [Yarwood 77].

The GUIDE debugger keeps an execution history that can reverse and replay the execution. Recording is done when the user selects a step forward function in the debugger view. When stepping backwards, the environment is restored to exactly the same state before stepping forward was initiated.

The above list is not complete. There are several other features that can enhance the usability of the debugger.

1. Ease of use

Even though a significant portion of software development time is spent in debugging, debuggers are seldom used [Bovey 87, Leintz 80, Panel Session 83]. One major obstacle to the use of conventional debuggers is that they usually have a command language which is complicated and difficult to remember. The complexity is due to the fact that the structure of the object under examination and manipulation, which is the state of a program, is in itself complicated. A careful design of the command syntax (e.g. dbx [dbx 83]) will help, but with the introduction of high-resolution graphics and mouse interface, other approaches would be possible [Bovey 87].

In the GUIDE debugger, all commands are carried out by using the same pop-up menu paradigm as the other five views. Textual information is required only during

editing of primitive structures (Section 2.3.1.1). There is no command language for the user to memorize.

2. Single-tool environment

Programmers typically use a loosely coupled set of tools in software development, such as an editor, compiler, linker, loader and debugger. These tools often have different command syntax and the user must switch to different tools between modifying a program and debugging it. Integrated programming environments are developed to improve the situation. Well known examples are Interlisp [Teitelman 81] and Smalltalk [Goldberg 84]. The tools in an integrated environment usually share some common internal structures that represent the program and provide a consistent user interface. Magpie [Delisle 84], an interactive programming environment for Pascal, takes a further step towards integration. It provides an environment that appears to the user as a single tool; there are "no firewalls separating the various functions provided by the environment" [Delisle 84]. The user is not aware of "being in the editor", "running the compiler" or "using the debugger". The most significant advantage of the single-tool environment is that it is easier and more pleasant to use. The user does not have to perform mental context switches when developing a program.

The GUIDE debugger view provides a syntax directed editor that supports incremental compilation. It presents a single tool for editing, compiling and debugging. The view is divided into subwindows, or panes, which display a complete picture of all aspects of the executing state of a program. The code pane has the same editing paradigm as the code view (Section 2.3) and therefore provides immediate feedback of any semantic error. Incremental compilation generates the statement tree without any involvement from the user. It also implies

that there is no need to re-compile the program or restart the debugger after modifications are made to the code.

3. Immediate execution

Immediate execution allows procedures or functions to be invoked directly without a calling program. Traditionally, a substantial part of a program must be written before any testing or debugging activity can begin. It is difficult to test the procedures of the program before integrating them together. This feature allows each procedure to be individually coded, tested and debugged. It also eliminates the need for many test procedures and therefore saves a lot of development time.

When a debugging session is started on a procedure, the GUIDE debugger itself initiates the procedure call. The user can supply the parameters to the procedure call by directly modifying their values in the data display pane.

It should be noted that the GUIDE debugger prototype has implemented almost all the aforementioned desired features while maintaining an interface which is easy to learn and easy to remember.

5.3 The Philosophy of Debugging

The features provided by the debugger view have a major impact on the way programmers would write a program. Typically, a program is written using a text editor. Then it is compiled. If any syntactic or semantic error is detected, it is corrected with the editor and re-compiled again. This process is repeated until no more static errors exist. The object code produced by the compiler is then linked with the library to produce an executable version. During the linking phase, duplicated or missing procedures may be discovered which will require the program to be edited, compiled and linked again. Eventually, the

program is ready to be tested and debugged. During testing, the intended behavior and the actual behavior of the program are compared and analysed to find the cause of variance. Usually, a substantial amount of the code has to be written before any testing can begin. Because of this, localizing the bugs is often a difficult task. A lot of time the programmer has to make a conjecture of where the bugs might be. When the bugs are located, the source code is edited again to implement the corrections. The entire cycle is repeated until a sufficient match between desired and observed behavior is obtained. In the process, many test procedures would have to be devised and then removed after testing is done. The test procedures themselves may create more bugs or hide existing ones which make the testing and debugging activities even more difficult.

Using the GUIDE debugger, the programmer will not face many of the problems in the above conventional approach. First, the syntax directed editing and semantic error handling mechanisms of the integrated editor guarantee the code to be syntactically and semantically correct. Then the incremental compilation feature eliminates a separate compiler. Because the code is interpreted, there is no need for a linker and a loader. Once a statement is entered into the code, it can be analysed, tested and debugged immediately. It is not necessary for the program to be completed before debugging can begin because immediate execution allows individual procedure to be invoked directly by the debugger. The design of the debugger encourages the user to perform coding and testing simultaneously and to debug each individual procedure before integrating it into a larger program. This new way of programming can uncover the bugs at an early stage before they are obscured by other things. Software development time can be reduced significantly with this approach.

5.4 The GUIDE Debugger View

Figure 5.2 shows a debugger view. The view is divided into six subwindows, or panes, which represents different aspects of the execution state of a program:

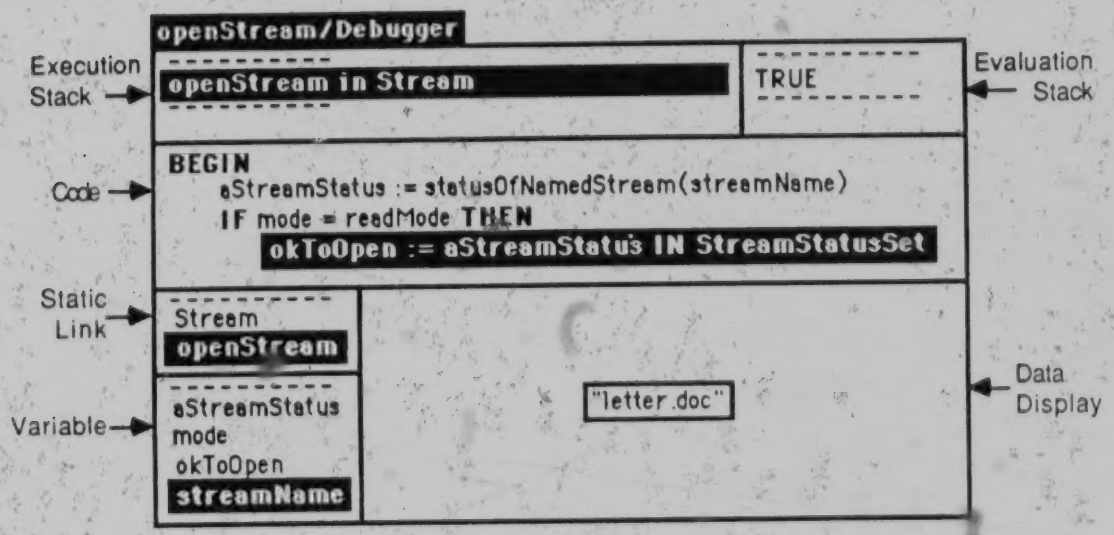


Figure 5.2 Example of a debugger view.

1. Execution stack

Modula-2, and other ALGOL-like languages, divide the program into procedures. A procedure is activated by another procedure. The execution stack displays what procedures the program has called, and in which order, to reach the current execution position. At execution time, when a procedure is called, an activation record is allocated for that procedure. The activation record contains all the information necessary to execute the procedure, including the storage space associated with the local variables of the procedure. Each item on the execution

stack is actually an activation record and thus the execution stack is in effect a representation of the dynamic chain. The most recently called routine is displayed on the top of the list, followed by its caller and the previous caller, all the way back to the context invoked by the debugger. The current executing state of any procedure on the execution stack can be examined by selecting it in this pane. All the other panes will be updated to display the current environment of that activation record. The menu provided in the execution stack pane is shown in Figure 5.3.

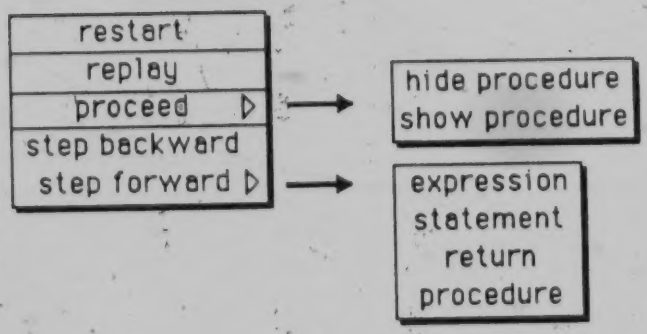


Figure 5.3 The execution pane menu.

The menu allows the user to control the flow execution by stepping forward and backward, and replay the execution history. Each of the menu item is explained below:

restart

Selecting **restart** command from the pop-up menu sets all local variables in the selected context to their uninitialized state and the execution to the start

of the context. All activation records above the selected one in the execution stack are removed.

replay

The **replay** command replays the execution history from the start of the debugging session to the current execution point. The flow of execution is replayed by highlighting the executing statements in the code pane and the variables are updated inplace in the data display pane.

proceed—hide procedure

The **proceed—hide procedure** command continues execution from the current point without stepping into procedure calls until one of the following events occurs:

- the context terminates,
- a breakpoint is encountered, or
- the user holds down the left shift key to interrupt.

As the program is executing, each statement is highlighted to animate the flow of execution and variables are updated so that the user can see the effect.

proceed—show procedure

Similar to the previous command, the **proceed—show procedure** continues execution but also traces into procedure calls.

step backward

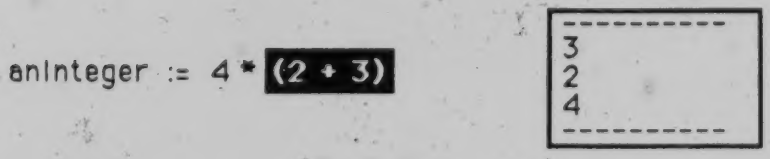
The **step backward** command allows the state of execution to be restored to the previous step. All variables that were changed since the previous step are restored to their previous values and the highlight bar in the code pane is set to the statement just before stepping has occurred. Stepping backward allows the user to back up a computation, change something, and then repeat the computation in the new environment.

step forward—expression

There are several ways of stepping forward, the first is stepping into an expression. Stepping into an expression reveals the flow of execution in a more complicated expression. For example, consider the assignment statement

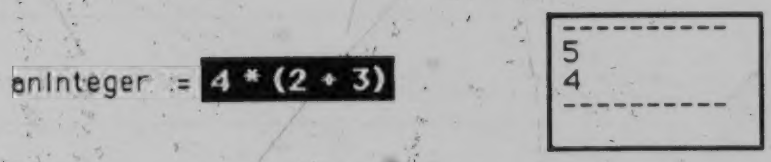
```
anInteger := 4 * (2 + 3)
```

The highlighting sequence of the expression and the contents of the evaluation stack are shown in Figure 5.4.

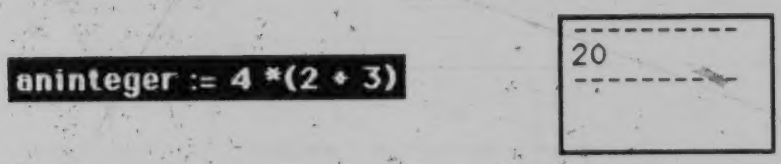


(a) Evaluating 2 + 3.

Figure 5.4 Expression stepping of anInteger := 4 * (2 + 3).



(b) Evaluating 4 * 5.



(c) Assigning 20 to anInteger.

Figure 5.4 Expression stepping of anInteger := 4 * (2 + 3). (cont.)

step forward—statement

The **step forward—statement** command completes the execution of the current statement in the code view and suspends execution at the next statement. Procedure calls will not be traced. To step into a procedure, the **step forward—statement** command must be chosen instead.

step forward—return

The **step forward—return** command continues execution at the current context until the current context is about to return to its caller. Procedure calls will not be traced.

This is useful in two circumstances: when the user has accidentally stepped into a procedure that he is not interested in; or when the user has determined that the current procedure works to his satisfaction, and he does not want to slowly step through the rest of it.

step forward—procedure

The **step forward—procedure** command continues execution until a procedure call statement is encountered; then the procedure is invoked and execution is suspended at the start of that procedure.

2. Code in execution

The code pane shows the code of the highlighted context. As the code is executed, the statement or expression to be evaluated next is highlighted. Thus the user can see the flow of execution dynamically and is able to interpret the changes in the execution environment.

The menu provided in the code pane is the same one as in the code view. The user is allowed to edit the code directly in the code pane without having to open another code view. Syntax and semantics checking are also carried out as in the code view. This makes it easier to fix logical errors that are discovered during debugging.

3. Evaluation stack

The debugger emulates a stack machine. When an expression is to be evaluated, the operands of a mathematical operation in the expression are pushed onto the evaluation stack. The debugger then pops the top elements on the stack, performs the operation on them, and pushes the result on the stack so that it will be available for use as an operand of the next operator. The actual parameters of a procedure

call are also pushed onto the stack in the same way. Upon return of the procedure call, the parameters are popped off the stack and the result of the procedure call, if any, is pushed onto the stack. An example of executing an expression is shown in Figure 5.4.

This feature allows the user to observe in detail the evaluation of an expression. Furthermore, any element on the stack can be modified at will. The user simply selects the element to be modified, and activates the menu by pressing the middle mouse button. A dialog box is opened for the user to enter the new value. A complete type checking will be performed to ensure the new value is compatible with the previous one. This feature is particularly useful in aiding the user to understand the evaluation of a complicated expression.

4. Static link

Modula-2 adopts static scoping rules to specify the scope of variables. Static scoping has been discussed in Chapter 4. The static link pane represents the static chain of a given context. Selecting a context in this pane will list the local variables of the procedure in the variable pane as explained below.

5. List of visible variables and constants

The list of variables and constants is presented in the variable pane when a particular activation record is selected in the static link pane. In the case of a procedure or function activation record, the list contains the local variables and constants declared in that procedure or function. For the activation record of a module, the list shows the global variables and constants declared in that module as well as the public variables and constants of all imported modules. Private variables and constants of an imported module are not displayed. This will

preserve the concept of abstraction provided in Modula-2 while allowing the user to view and modify all visible variables. The imported variables and constants are made distinct by qualifying them with the imported module name.

The menu provided in this pane has only one item, **view**, which will open a separate window for the selected variable or constant. This new window, the inspector window, has two panes: the left pane shows the variable name and the right pane shows the value of the variable, as shown in Figure 5.5. The right pane has a vertical scroll bar as well as a horizontal scroll bar since the data display is two-dimensional. If the user holds down the left shift key and press the left mouse button, the cursor will change into a cross as shown. This indicates that grab mode is in effect: the user can grab the picture and move it two-dimensionally.

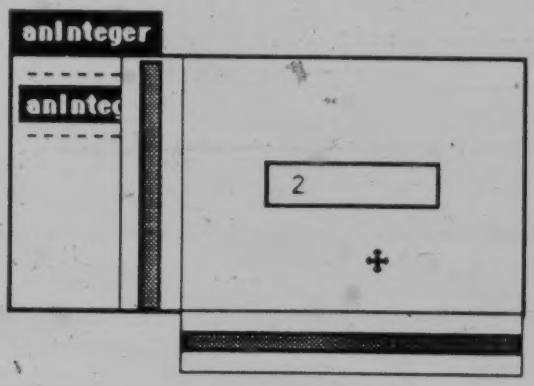


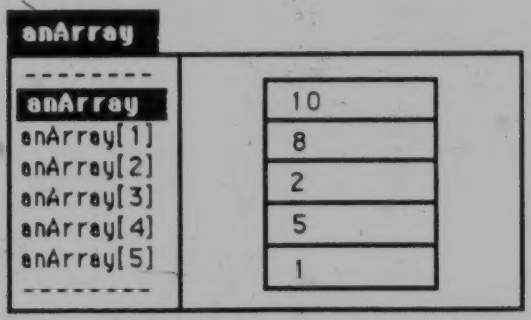
Figure 5.5 Example of an inspector view.

In the case of a more complex data structure, such as an array, the left pane also lists all the sub-components of the data structure, and the right pane will display the value of the sub-component if one is selected. Figure 5.6 shows the window opened for a one-dimensional array. The left pane lists the array name and all the

elements of the array. When the array name is selected, the entire array is displayed in the right pane as shown in Figure 5.6a. When an element is selected, only the value of the element is displayed (see Figure 5.6b).

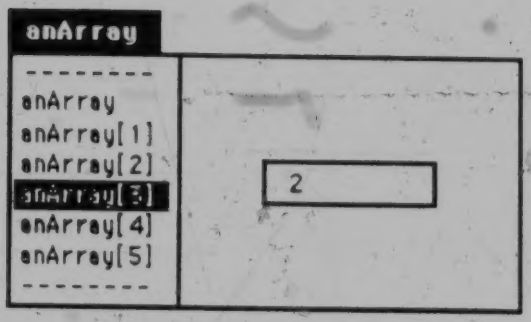
The left pane of the window has the same menu as that of the variable pane in the debugger view and it works exactly the same way. Thus the user may open more inspector windows for the subcomponents of a variable. For example, Figure 5.6c shows several overlapping inspector windows opened for different elements of the array. Each window can be reduced to a smaller size to eliminate overlapping. In the case of a multi-dimensional array or any complicated data structure, this feature allows the user to trace the data in the data structure to any degree of detail.

When the program under examination is being executed, all the inspector windows, if affected, are updated dynamically. For example, during the debugging session of a sorting program, the user may open an inspector window on the array being sorted, and "see" the sorting process taking place.

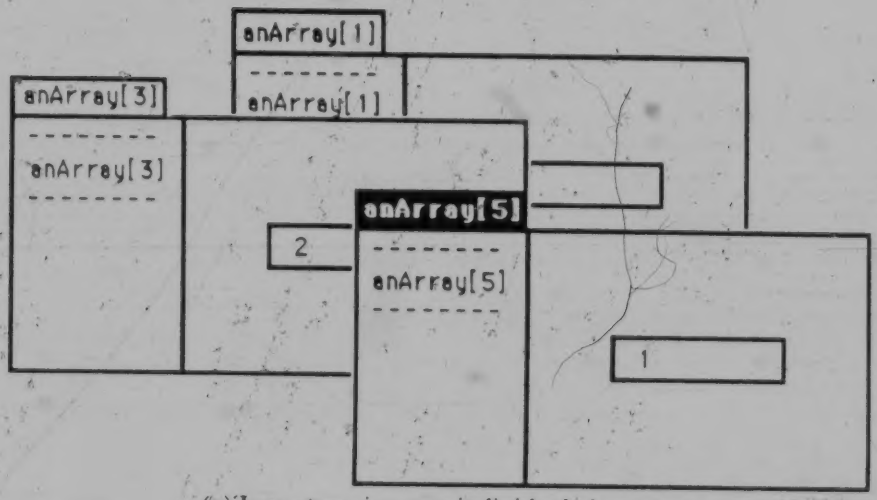


(a) Displaying the entire array

Figure 5.6 An inspector view on an array.



(b) Display element anArray[3]



(c) Inspector views on individual elements

Figure 5.6 An inspector view on an array. (cont.)

6. Pictorial display of data structures

Instead of displaying the data structures in a textual manner, the debugger displays them pictorially in this data display pane. There are two major considerations in a pictorial display scheme: the appearance of the display and the location of the display. Currently, all data structures are displayed in a predefined format. Figure

5.7 shows the default formats for all of the basic types and the formats for the composite data structures.[1]

The menu in this pane provides an editing feature. When a data display of a variable is selected, the menu can be invoked to edit the data. A dialog box is opened for the user to enter the new value for the variable as shown in Figure 5.8 and a complete assignment compatibility check is done before the debugger accepts the new value.

A debugging session is ended by closing the debugger view. The user may abort the debugging session any time before the execution of the program is completed. Any inspector window opened during the debugging session is also closed automatically when the debugger view is closed.

---

[1] Currently, pointers are not supported yet.

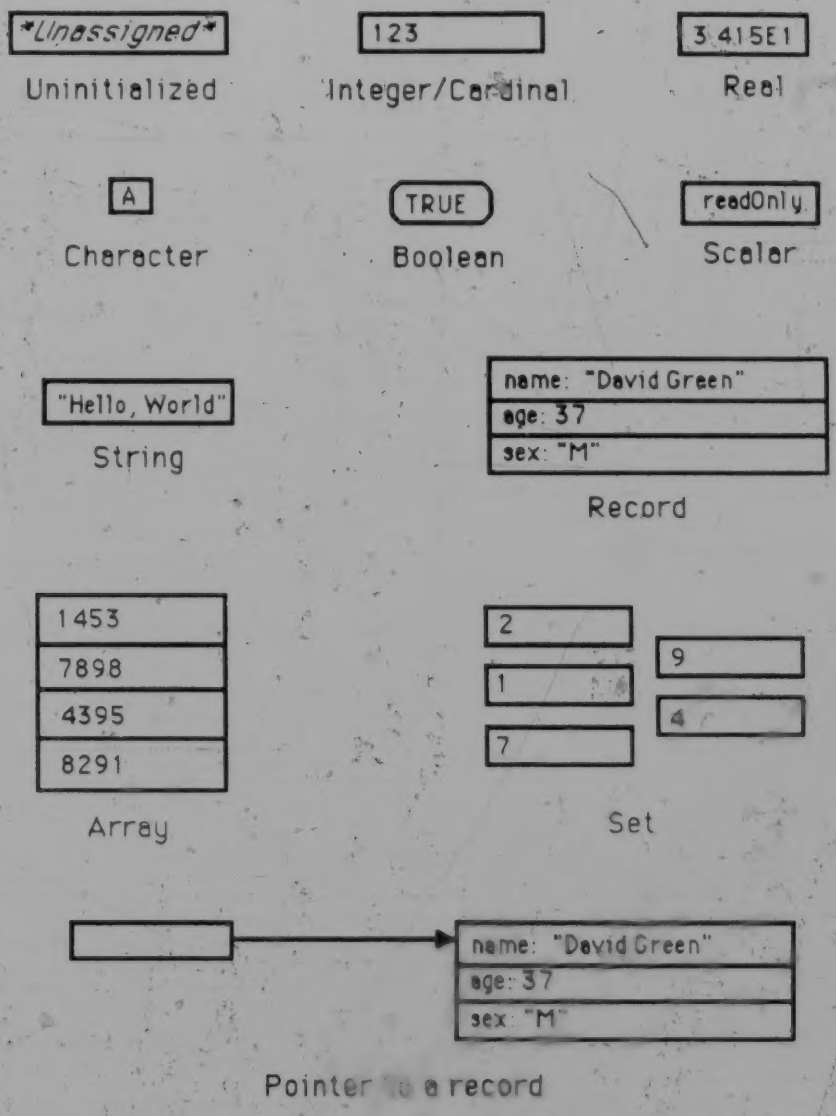


Figure 5.7 Display format of data structures.

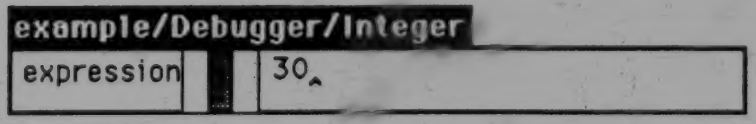


Figure 5.8 A dialog box for editing a variable.

CHAPTER 6

IMPLEMENTATION OF THE GUIDE DEBUGGER

This chapter describes the implementation of the GUIDE debugger. Besides the functions provided by a conventional debugger, the GUIDE debugger is able to track the history of execution and to display the data structures pictorially. We will first present the implementation of the pictorial display of data structures. Then we will show how single stepping is performed by the debugger and how execution history is recorded. Following this is a description of the user interface of the GUIDE debugger. Lastly, we will present how testing was performed on the debugger.

6.1 Pictorial Display of Data Structures

We have discussed how GuideStorage allocates storage space in Section 4.6. GuideStorage also has the ability to display itself graphically. It implements the display method:

```
displayOn: aDisplayMedium at: aDisplayPoint clippingBox: clipRectangle
        rule: ruleInteger mask: aForm
```

This is a standard Smalltalk display message understood by all graphical objects in the system. It displays the receiver on the display medium, aDisplayMedium, positioned at aDisplayPoint within the rectangle clipRectangle and with the rule, ruleInteger, and mask, aForm. Pixels that fall outside of the clipping rectangle are not displayed. The display message is usually sent by the inspector view. In which case aDisplayMedium would be the screen, clipRectangle would be the rectangular area of the view, aDisplayPoint is computed so that the picture would be centered within clipRectangle, ruleInteger would overwrite whatever is on the screen and aForm would use a black outline for the display.

6.1.1 Display of Primitive Data Structures

GuideLiteralStorage's method is very simple:

GuideLiteralStorage

```
displayOn: aDisplayMedium at: aDisplayPoint clippingBox: clipRectangle
rule: ruleInteger mask: aForm
    "Display the pictorial representation of myself."

    self form
        displayOn: aDisplayMedium
        at: aDisplayPoint
        clippingBox: clipRectangle
        rule: ruleInteger
        mask: aForm
```

The message self form returns a graphical object that is a representation of the content of GuideLiteralStorage. Currently, the display is simply some text within a box as shown in Figure 5.7. Future implementation may allow the user to define his own display routine. The instance variable display inherited from GuideStorage is used to cache this graphical object so that it does not have to be re-created every time. When the content of the storage is modified, the display is updated to reflect the changes.

6.1.2 Display of Composite Data Structures

The subcomponents of a composite data structure are also instances of GuideStorage. The display is determined by two pieces of information:

- the graphical representation of each subcomponent, and
- the relative location of the subcomponents.

GuideCompositeStorage provides the abstraction to display the entire data structure and its subclasses implement the methods for creating the pictures and specifying the display locations for the subcomponents.

GuideCompositeStorage

```
displayOn: aDisplayMedium at: aDisplayPoint clippingBox: clipRectangle
rule: ruleInteger mask: aForm
      "Display the pictorial representation of myself."

      1 to: elements size do:
            [:index |
            (self displayElement: index)
                  displayOn: aDisplayMedium
                  at: aDisplayPoint + (elementBoxes at: index) origin
                  clippingBox: clipRectangle
                  rule: ruleInteger
                  mask: aForm]
```

The message (self displayElement: index) returns the graphical objects which represent the subcomponents and elementBoxes is an array of rectangles specifying the locations and sizes of the subcomponent displays.

For example, GuideRecordStorage implements the method displayElement: by appending the field name with the graphical representation of the field. The elementBoxes also specify that the fields are to be displayed in a stack format as shown in Figure 5.7.

The graphical representation of a subcomponent is provided by the subcomponent itself. It is up to each individual subcomponent to decide how to display itself as part of the picture of a composite data structure. Therefore it is not required to know what are the types of the subcomponents in order to display them. Because the subcomponent may be another composite data structure, subclasses of GuideCompositeStorage must convert its more complicated display into a form that is appropriate to be displayed within another composite data structure.

Limited workspace is often a problem when displaying large data structures. The data display pane provides a vertical scroll bar and a horizontal scroll bar. Unlike a paragraph of text which can be wrapped around, a pictorial display is two-dimensional and therefore

the scrolling must also be two-dimensional. This scrolling mechanism provides an unlimited workspace for data structure display.

6.1.3 Display of Pointers

The most difficult displays are data structures that involve pointers such as linked lists and trees. In other data structures such as records and arrays, the locations of the subcomponents are fixed relative to the location of the aggregate structure. With pointers however, that is not the case. There are two decisions to be made. First, how the arrows should be drawn from the pointer to the referent and secondly, where the referent should be displayed relative to the location of the pointer. The decisions are both technical and psychological. For example, consider trees. The user may view the tree as an inverted tree or as a doubly linked list (Figure 6.1). The debugger has no way of knowing which way is preferred by the user unless explicit instructions are given. Furthermore, as the tree grows, the locations of the existing nodes must be adjusted to allow more room for new nodes. This operation can be both complicated and time consuming. Since the current implementation of GUIDE does not support POINTER types, the GUIDE debugger is not able to incorporate the display of pointers.

(a) Doubly linked list (b) Inverted tree

Figure 6.1 Two display methods of tree.

Myers [80, p.40 - 43] has devised a simple and efficient algorithm to display pointers. His algorithm may be used as a starting point for GUIDE. Incense uses *layouts* to specify where pointer-containing data structures should place their child nodes. Basically, a layout divides a rectangular area into several smaller ones. One subarea is for the pointer-containing data structure, and one subarea for each object pointed to. For example, for a record containing 2 pointers, a layout would divide the display area into 3 pieces: one for the record and one for each of the two referents (see Figure 6.2).

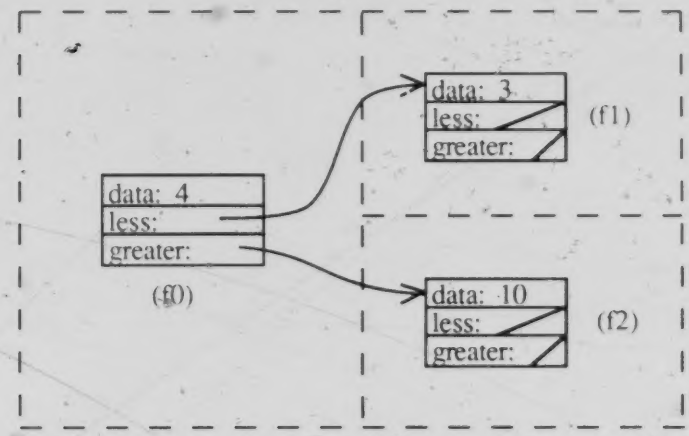


Figure 6.2 Layout with 3 fields: one for record: (f0), and one for each referent: (f1) and (f2). Dotted lines are added to show the area of each field. Adopted from [Myers 80].

In a recursive data structure such as tree, there are layouts at each level of nesting. They get progressively smaller since the area provided for them is reduced at each level. This theoretically would allow the display of an arbitrary number of levels, but, in fact, after a threshold is reached and the nodes are too small to see, displaying terminates (see Figure 6.3). Pointers to data structures that are already displayed do not cause an infinite cycle, since an arrow is simply drawn to the original occurrence as shown in Figure 6.4. Splines

are used to display the arrow lines rather than using straight lines so that they do not cause confusion with other lines in the pictures (Figure 6.5).

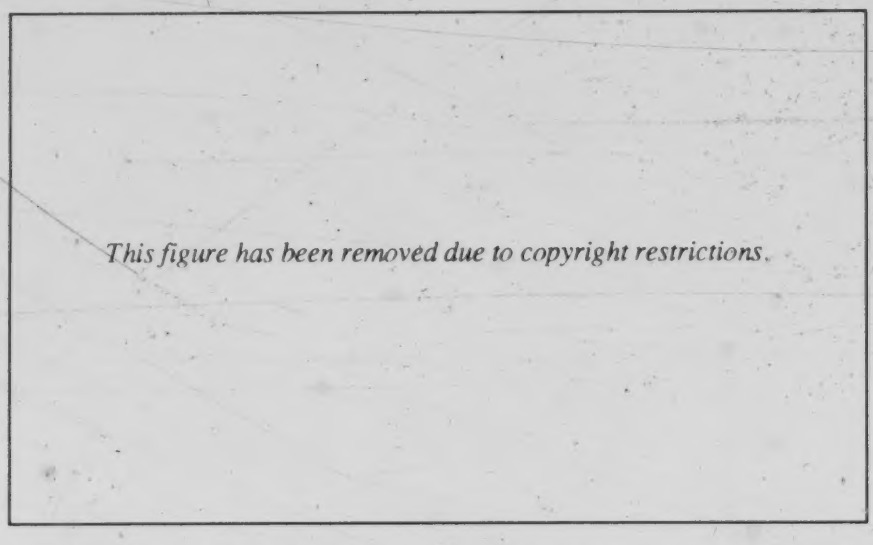


Figure 6.3 Deep recursive tree display demonstrating how elements get smaller. Overall structure, however, is easily understood [Myers 80].

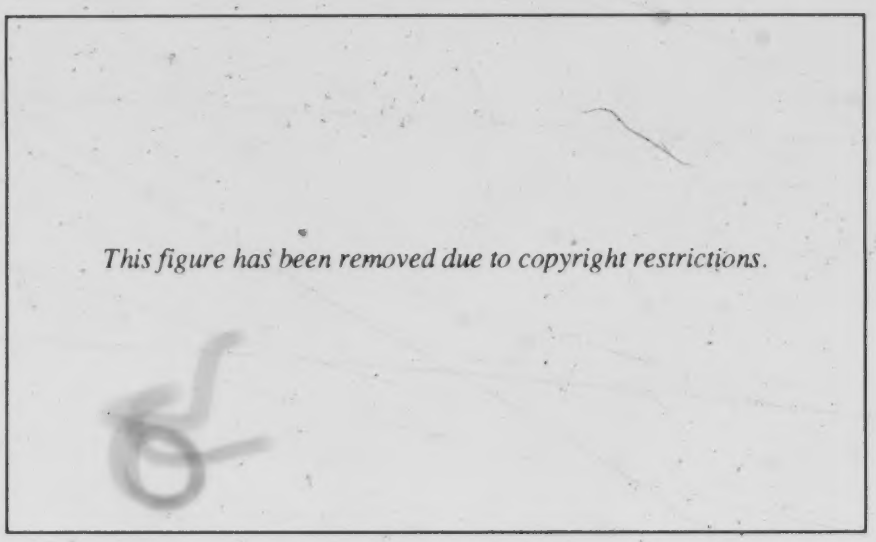


Figure 6.4 Pointer to previously displayed object does not generate a new copy. The second arrow is drawn to the first occurrence [Myers 80].

Figure 6.5 Demonstration of the advantage of curved lines used in Incense (a) over straight lines (b). The knots used in drawing the spline are shown as black squares in (a) [Myers 80].

6.1.4 User-defined Display Routines

The current implementation of GUIDE does not allow for user-definable display routines. However, this is a desirable feature because the default display format may not suit individual need. When defining a data type, the user may specify a display method for that type. All variables declared for that type would then be displayed according to the instructions given by the user. To make provision for this, an instance variable, say `userMethod`, should be added to `GuideDeclarationSymbol`, which is the superclass of all data type symbols. The `userMethod` specifies the name of a user-defined display method. The display method must be implemented under the appropriate subclass of `GuideStorage` for the data type. GUIDE must provide an interface to do this so that the user does not have to find out which subclass to use. This can be done easily because all subclasses of `GuideDeclarationSymbol` have already implemented the method `storageClass` which returns the name of the appropriate storage class. Also, Smalltalk provides a way to "spawn" an editing window which contains all the methods defined under a specified category and allows the user to define new ones. The original display method must be changed. For example, `GuideLiteralStorage`'s display method may be changed to something like this:

GuideLiteralStorage

```
displayOn: aDisplayMedium at: aDisplayPoint clippingBox: clipRectangle
rule: ruleInteger mask: aForm
        "Display a graphical representation of myself."

        | displayMethod picture |
        displayMethod ← type userMethod.
        displayMethod == nil
                ifTrue: [picture ← self form]
                ifFalse: [picture ← self perform: displayMethod].
        picture displayOn: aDisplayMedium
                at: aDisplayPoint
                clippingBox: clipRectangle
                rule: ruleInteger
                mask: aForm
```

It first obtains the name of the user-defined routine from the type of the data structure. If the user did not specify any display method, then the default (form) is used. The Smalltalk system message perform: is sent to GuideLiteralStorage to execute the user defined method which should return a graphical object to be displayed. GuideCompositeStorage can follow the same idea to accommodate for user-defined routines.

This approach has a drawback. The user is required to learn Smalltalk in order to define his own display routines. A better way is to allow the user to define the routine in the same source language, i.e. Modula-2. A set of Modula-2 callable routines must be provided to create and manipulate graphical objects. To improve performance, GUIDE may translate the Modula-2 code into equivalent Smalltalk code. This places heavy demand on GUIDE but achieves a better user interface without speed degradation.

6.2 Single Stepping of Execution

The debugger provides four ways to step forward: **step expression**, **step statement**, **step procedure** and **step return**. The actions performed by each function are fully explained in Section 5.4. Figure 6.6 shows the flow charts of these functions.

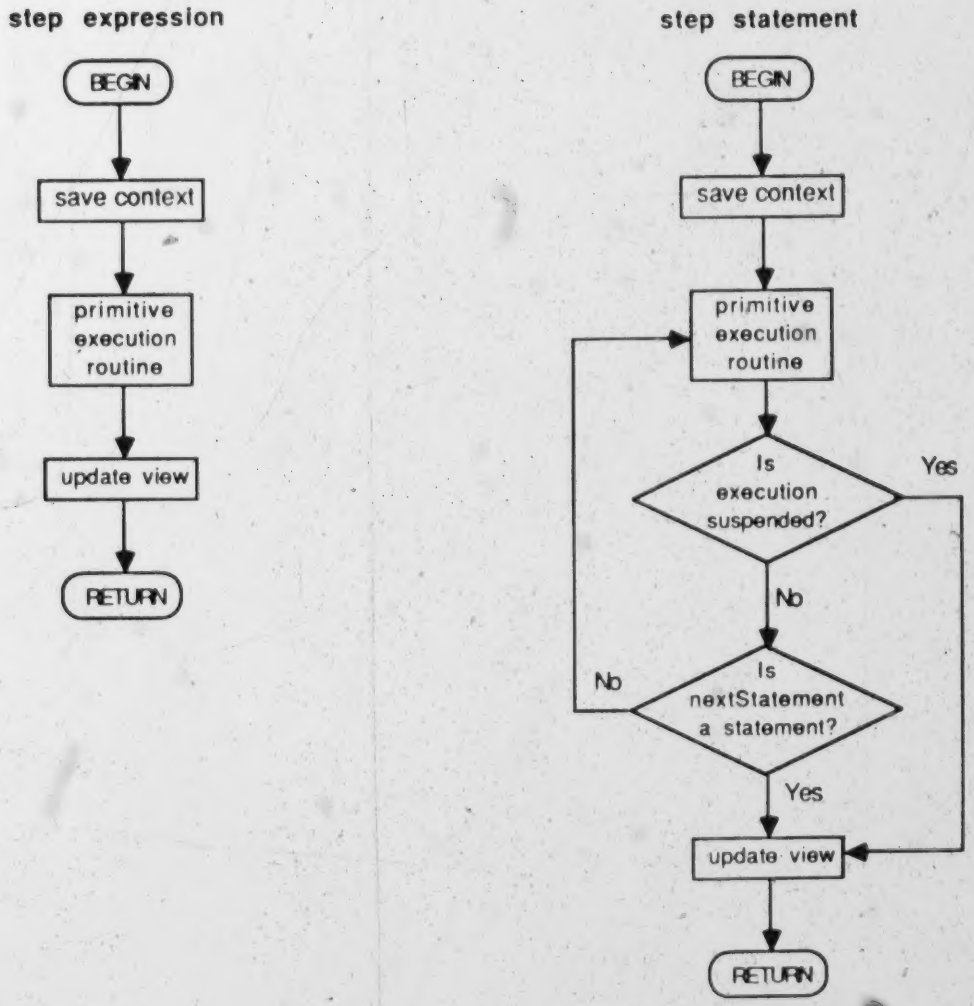


Figure 6.6 Flow charts of stepping forward functions.

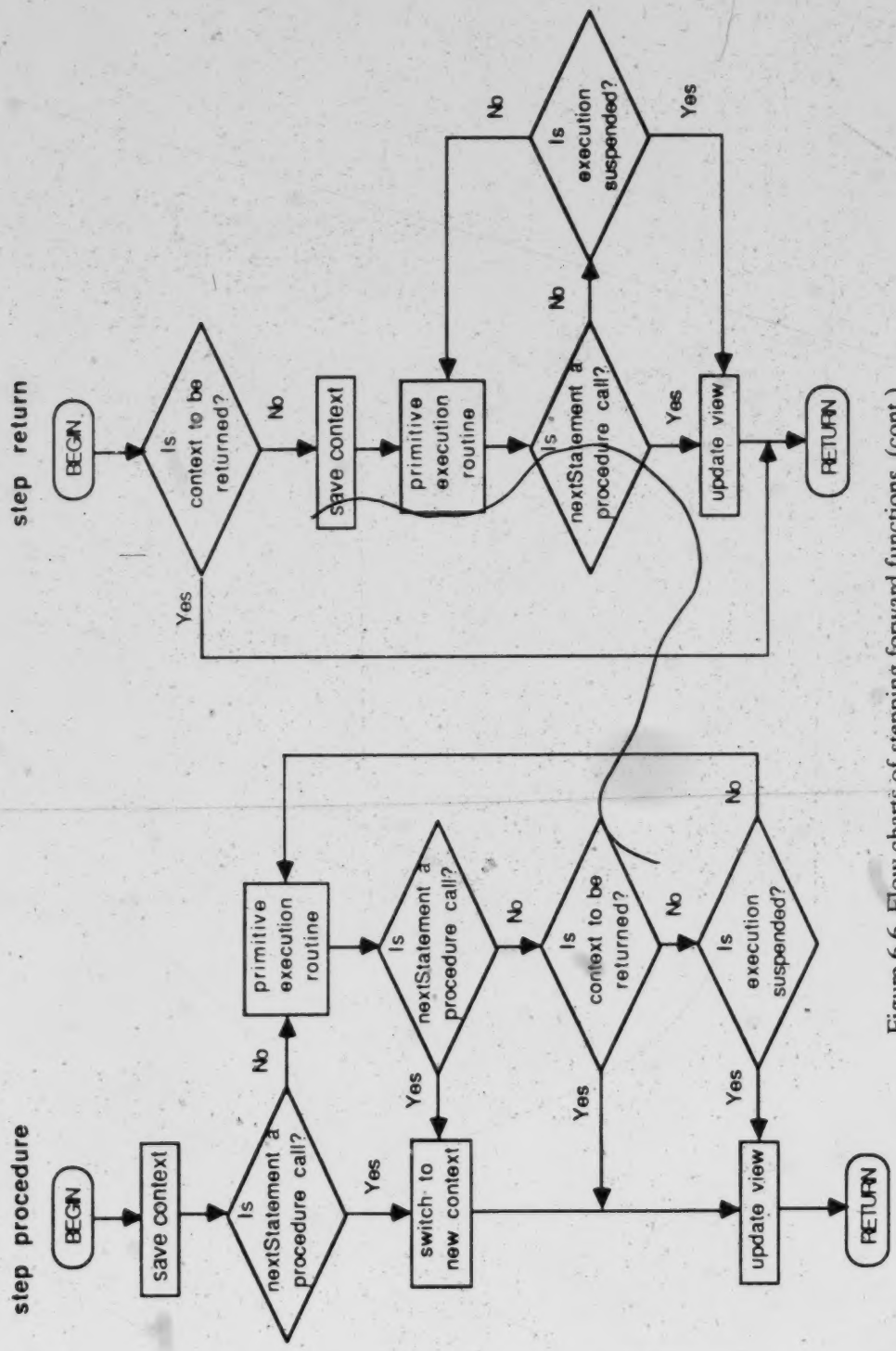


Figure 6.6 Flow charts of stepping forward functions. (cont.)

In all cases, the primitive execution routine is called to perform the stepping. The primitive execution routine is implemented by ActivationRecord as discussed in Section 4.7. It should be noted that history recording is initiated by the debugger and is done before stepping is carried out. During the actual execution, there is no recording of any statement being executed. Therefore it allows the user to reverse the execution to a previous stepping point.

Except for step procedure, all other stepping functions do not trace into a procedure when a procedure call statement is encountered, but will complete the execution of the call. However, during execution of the procedure call, the activation record can be placed into a suspended state. This can happen in three situations: 1) a runtime error has occurred, 2) a breakpoint is encountered, and 3) the user has pressed the left shift key to pause the execution. When suspension occurs, the debugger view must be updated to display the new environment where execution is stopped. After the debugger returns control to the user, he may correct the runtime error or specify a debugging action.

Step expression simply calls the primitive execution routine once and then returns. It stops after each parse node is executed and therefore allows the user to trace into the evaluation of expressions. **Step statement** keeps calling the primitive routine until the next parse node is a statement. Every parse node understands the message isStatement and answers whether it is a statement or not. GuideExpressionNode and its subclasses return false since they do not represent Modula-2 statements; all other parse nodes answer true. Therefore, **step statement** does not stop at places such as binary expressions, but stops before a statement is to be executed. **Step procedure** continues the execution until a procedure call is encountered or until the current activation record is about to return to its caller. **Step return** keeps executing until the context is about to return to its caller.

6.3 Execution History

A unique feature of the GUIDE debugger is its ability to step backward and to replay. This is accomplished by keeping a history of the execution. Every activation record has its own history. The instance variable history of ActivationRecord is an instance of class History. Class History has several instance variables as listed below. History provides two stacks to record any changes in the environment and two index pointers to replay the execution history.

activationRecord	a back pointer to the activation record whose execution is recorded.
historyStack	a stack to record changes in the environment.
valueStack	a stack to record the contents of the evaluation stack.
index	an integer index pointing to an element on historyStack to be replayed.
valueIndex	an integer index pointing to an element on valueStack to be restored on the evaluation stack.

A snapshot of the state of the activation record is taken whenever the user chooses to step forward. The next statement to be executed is saved on the historyStack and the contents of the evaluation stack are saved on the valueStack. Then the execution is stepped forward. During the execution, a number of statements may have been executed but only the first statement is recorded when stepping forward is initiated. When the user wants to step backward, the last statement being recorded is restored from historyStack and the contents of the evaluation stack are restored from valueStack. In effect, the activation record is restored to its previous state just before the stepping occurs. Since the user usually performs stepping forward and backward around a section of code that is of particular interest, the debugger essentially records "points of interest" rather than the entire

execution history. This will reduce the memory requirements significantly. Because history recording is initiated by the debugger view, it is possible for the debugger to disable any recording if the user decided not to step backward. This disabling feature is not implemented in the current version but can be incorporated easily.

The assignment statement, the FOR statement and the user all have the ability to modify the state of memory storage. Modifications by the user can occur at any time and are not controlled by the flow of execution. Therefore keeping a history of the flow of execution alone cannot determine when memory modifications have taken place. Only a memory storage cell (an instance of GuideStorage) knows when it is being modified. The best way to resolve this problem is to have each storage cell keeps a history of its modifications. When a storage cell is modified, it saves its old value on an internal stack and also sends a message to the current active activation record to record the change. The current activation record in turn passes the message to its history and history saves a pointer to the storage cell on its historyStack. The current activation record is the one at the top of the dynamic chain. It may not be the same one that contains the storage cell. By keeping the changes in the current activation record, the order of events that have taken place is properly recorded. To step backward, the history removes an item from its historyStack. If the item is an instance of GuideStorage, the storage cell is requested to restore its previous value. The history continues to remove items from the historyStack until the item removed is an instance of GuideParseNode. At this time the contents of the evaluation stack are also restored from the valueStack. Thus, the state of the environment is completely restored to the last execution break point.

Finally, the historyStack also contains instances of History itself. When a procedure call is made, the procedure call statement is first saved on the historyStack. The procedure is then activated. Before control is passed to the new activation record, the caller's history

also saves the history of the new activation record on the historyStack. Then the new activation record takes over control and saves changes on its own history until it completes execution and returns to the caller. The caller then continues execution. During backward stepping, if the top item on historyStack is an instance of History, restoration is passed to that history.

If historyStack becomes empty, then the restoration process must be passed to the caller's history. If the activation record's caller is undefined, it means the environment has been restored to the point where it started and no further backward stepping is possible.

The implementation of history replay is straightforward. All memory cells have to be reset to their uninitialized state before replay starts. History uses two instance variables, index and valueIndex, to keep track of the replay process. They are initialized to point to the first elements on the historyStack and the valueStack. The history then replays each item on the historyStack and increments index until such time as index reaches the top of the historyStack. At the same time, if the item on the historyStack is an instance of GuideParseNode, the valueStack is also restored and valueIndex is incremented. If the item is an instance of GuideStorage, the storage cell is requested to replay itself. GuideStorage also uses a stack index to identify which value on its internal stack is being replayed. The storage cell is set to that value and then the stack index is incremented to point to the next value for the next replay.

6.4 The User Interface of the GUIDE Debugger

The GUIDE debugger adopts the MVC paradigm from Smalltalk-80, just like the rest of GUIDE. The browser class in this case is the debugger itself and the internal structure is the activation of a procedure. Recall that a procedure has two components: a code segment

and an activation record. Therefore the debugger must provide functions to view and manipulate both components.

The debugger has a code pane used for editing the code (Figure 5.2). This is exactly what the code browser does. Defining the debugger to be a subclass of the code browser allows the debugger to inherit all the code editing functions. All that is required is to implement the functions to manipulate the activation record. During execution, the instance variable currentStructure inherited from StructureView is set to point at nextStatement of the active activation record. Since the code pane will always highlight currentStructure, the effect is a moving highlight bar on the executing statement.

The execution stack pane displays the list of activation records along the dynamic chain. It is constructed by following the caller instance variable of the activation record. When the user selects an activation record from this pane for inspection, the debugger broadcasts an update message to all the other panes. Each pane receives the update message and displays the appropriate information for the current selected activation record.

The evaluation stack pane displays the contents of the instance variable stack of the selected activation record. In the static link pane, the static chain for the selected activation record is listed. The static chain is found by recursively following the instance variable scope. Selecting an activation record in this static link pane displays the list of local constants and variables declared within that activation record in the variable pane. This list is in fact the keys of the dictionary valueTable of the activation record. Finally, the data pane displays a picture of the data structure selected in the variable pane. The pictorial display is obtained from the storage of the data structure as described previously.

6.5 Testing of Debugger

To test the runtime environment and debugger, four Modula-2 programs were written to be executed by the debugger. Each performed test on a different aspect:

- variable updates and data structure display,
- execution flow control,
- context switch, and
- import and export capabilities.

In the first program, we declared variables for every Modula-2 type including some complicated ones such as two-dimensional array and array of records. The program consists of assignment statements to every declared variables and also expressions that made references to variables and their subcomponents, such as array elements and record fields. A debugging session was started on this test program to observe variable updates and data structure displays.

The second program was written to test the flow control. It has every Modula-2 statements and a variety of combinations such as **IF** statement within a **WHILE** loop. This allows us to test the execution and flow control implemented by each parse node.

The third program was the classic recursive function that returns the factorial of an integer. Because the program made references to the same variable in different activation records of the same function, it checked whether context switching was done properly.

The last program actually composed of two procedures declared in two modules. The first module exported its test procedure along with some variables and constants. The variables were initialized in the main body of the module. The other module imported these symbols and made references to them. This test program tested whether the main bodies of the

modules have been initialized properly and also tested the implementation of the import and export features.

6.6 Summary

The GUIDE debugger is implemented as a subclass of the code browser so that it inherits all the code editing methods. It adds the methods to control the flow of execution and to display the data structures pictorially. Pictorial display of data structures is performed by the storage cells which are instances of the subclasses of GuideStorage. Each type of data structure has a corresponding storage class so that each will be displayed differently. Stepping forward of execution is implemented by repeatedly requesting the activation record to execute its primitive execution routine. The class History is used to record the changes in the environment including the flow of execution and storage locations. The modification history of the storage cells is maintained by the storage cells themselves so that modifications by the user are recorded properly.

CHAPTER 7

RECOMMENDATIONS, SUMMARY AND CONCLUSION

7.1 Recommendations

1. Standardization of data type display

There is no universal standard as to how a particular data type should be displayed. A standard would help the user to quickly identify the type of a data structure from the shape of its display. The lack of a standard will confuse users as they move from one graphical display system to another.

2. Debugging Concurrent Processes

Modula-2 provides a module called Processes which offers the necessary facilities for multiprogramming at a high level of abstraction. Because of the time-dependent property of concurrent processes, it is difficult to debug concurrent programs. Often, the errors cannot be reproduced because of timing differences. "Difficulty in reproducing events makes it hard to determine the sequence of events that produced the error — debugging is a serious problem!" [Gehani 84]. However, the execution history kept by the debugger can easily reproduce and locate the errors. Currently the debugger does not support concurrent processes. Extending the capability of the GUIDE debugger to include concurrent processes would be of great value in such situations.

3. Moving to Production

The current version of GUIDE is implemented in Smalltalk-80. To put GUIDE and its debugger into production, it should be implemented in some other more efficient language. In recent years, object-oriented programming has received more attention and is becoming more popular especially in the world of personal computers. Two of the most popular languages, C and Pascal, already have object-oriented concepts built on top of them. Object-oriented languages appear suitable for implementing GUI environments. Rewriting GUIDE in a compiled object-oriented language has obvious advantages. Much of the design and architecture can be retained and the speed could greatly improve.

7.3 Summary and Conclusion

We have described the runtime environment and debugging in GUIDE. GUIDE uses two structures to simulate the runtime behavior of Modula-2 — the statement trees and the activation records. The activation record controls the execution by repeatedly sending messages to each parse node to execute itself and to determine the flow of execution. The object-oriented implementation of GUIDE eliminates the need for a separate code interpreter.

The debugger animates the flow of execution by highlighting the statements during execution. Users can step through the code, even stepping into the components of an expression. The effective use of windows and menus allow users to manipulate an executing program without using a complicated command language. Furthermore, the debugger provides some innovative features — the ability to step backward, to replay the execution history and to display the data structures pictorially.

Pictorial display of data structures is a useful feature. For example, an array is displayed in the way as a programmer perceives it. This form of display is more acceptable to human comprehension than textual information. The user is allowed to modify the values of the data structure directly in the display. Also, when the data structure is assigned a new value during execution, the display is updated immediately, creating an animation effect. The user can see the effect of execution, such as the sorting of an array. This can be an effective educational tool.

Integrating the debugger with a syntax directed editor that supports incremental compilation is found to be a powerful debugging feature. There are several implications as a result of this:

1. Single-tool environment

   It presents to the user as a single tool to develop programs. Usually, different tools are used for program modification, compilation and debugging. The user is often required to learn different command languages and perform mental context switches to use these tools when writing a program. Providing these functions within one environment eliminates the barrier that typically exists between different software development tools.

2. Syntax error elimination

   The syntax directed editor prevents the user from entering code that contains syntax error. By inspecting the statement tree, GUIDE generates context sensitive menus that consist of allowable items only. Therefore it is not possible fo the user to insert illegal expressions or statements into the program.

3. Immediate feedback of semantic error

   Incremental semantic analysis is performed as the user is coding the program. Any semantic error can be detected immediately and reported to the user. Only semantically correct information is allowed in the program.

4. Immediate execution

   Because the code is guaranteed to be free of static errors and the statement tree is already generated through incremental compilation, the program can be executed immediately. Many test procedures can be eliminated because procedures can be invoked directly by the debugger without a calling program. Individual procedure can be fully tested before it is integrated into a larger program.

5. Elimination of a separate compilation process

   Incremental compilation modifies the statement tree as the user is editing the program. The changes take effect immediately and therefore the user does not need to recompile the program after modifications are made. Because the values of the variables in the context being debugged remain unchanged, the debugging activity can be continued without having to restart the debugger.

Debugging is often considered as a separate process from coding. Programmers usually code a substantial part of a program first, then compile and test it. If errors are found, the program is modified, recompile and test again. This cycle continues until the observed behavior and expected behavior of the program match closely enough. The features provided by the GUIDE debugger change this traditional way of programming. It combines coding and debugging into one process and eliminates a separate compilation process. When coding a program, debugging activities can be carried out simultaneously. The user is also encouraged to develop and test the procedures first before integrating them

into a program. The reverse execution feature of the debugger is useful for analyzing a section of code and locating errors. Users can step forward and backward around the code and make corrections until the desired effect is achieved. This new philosophy of debugging is much more efficient.

The GUIDE debugger has demonstrated the effective use of GUI in software development. It combines many innovative features while maintaining a simple user interface. A software development tool like the GUIDE debugger will certainly help to reduce development and maintenance costs.

References

[Adams 86] Adams, E., and Muchnick, S.S., "Dbxtool: A Window-based Symbolic Debugger for Sun Workstations," *Software— Practice and Experience*, Vol. 16, 1986, pp. 653-669.

[Baecker 68] Baecker, R.M., "Experiments in On-Line Graphical Debugging: The Interrogation of Complex Data Structures, " (Summary only) *First Hawaii International Conference on the System Sciences*, January 1968, pp. 128-129.

[Baecker 75] Baecker, R.M., "Two Systems which Produce Animated Representations of the Execution of Computer Programs," *SIGCSE Bulletin*, Vol. 7, No. 1, February 1975, pp. 158-167.

[Baecker 81] Baecker, R.M., *Sorting out Sorting*, 16mm color, sound film, 25 minutes, Dynamics Graphics Project, Computer Systems Research Institute, University of Toronto, Toronto, Ontario, Canada, 1981.

[Baecker 83] Baecker, R.M., and Marcus, A., "On Enhancing the Interface to the Source Code of Computer Programs," *Human Factors in Computing Systems: Proceedings SIGCHI'83*, Boston, MA, December 1983, pp. 251-255.

[Beizer 84] Beizer, B., *Software System Testing and Quality Assurance*, Van Nostrand Reinhold, New York, 1984.

[Boehm 73] Boehm, B.W., "Software and Its Impact: A Quantitative Assessment," *Datamation*, Vol. 19, No. 5, May 1973.

[Bovey 87] Bovey, J.D., "A Debugger For A Graphical Workstation," *Software — Practice and Experience*, Vol. 17, No. 9, September 1987.

[Brown 84] Brown, M.H., and Sedgewick, R., "A System for Algorithm Animation," *Computer Graphics: SIGGRAPH'84 Conference Proceedings*, Minneapolis, Minn., Vol. 18, No. 3, July 1984, pp. 177-186.

[Brown 85] Brown, G.P., Carling, R.T., Herot, C.F., Kramlich, D.A. and Souza, P., "Program Visualization: Graphics Support for Software Development," *IEEE Computer*, Vol. 18, No. 8, August 1985, pp. 27-35.

[Cargill 84] Cargill, T.A., "Debugging C Programs with the Blit," *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, 1984, pp. 1633-1647.

[Cargill 85] Cargill, T.A., "Implementation of the Blit Debugger," *Software—Practice and Experience*, Vol. 15, No. 2, 1985, pp. 153-168.

[dbx 83] "DBX(1)," *UNIX Programmer's Manual*, 4.2 Berkley System Distribution, Vol. 1, Computer Science Division, University of California, Berkley, CA, August 1983.

[Delisle 84] Delisle, N.M., Menicosy, D.E. and Schwartz, M. D., "Viewing a Programming Environment as a Single Tool," *Proceedings of ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environment, SIGPLAN Notices*, May 1984, pp. 49-56.

[DOS 88] *Microsoft MS-DOS Operating System Version 4.0 User's References*, Microsoft Corporation, WA, 1988, 430 pp.

[Eisenstadt 89] Eisenstadt, M., and Brayshaw, M., "AORTA Diagrams as an Aid to Visualizing The Execution of Prolog Programs," *Graphics Tools for Software Engineers*, Cambridge University Press, Cambridge, UK, 1989, pp. 27-45.

[Feldman 89] Feldman, M.B., and Moran, M.L., "Validating a Demonstration Tool for Graphics-Assisted Debugging of Ada Concurrent Programs," *IEEE Transactions on Software Engineering*, Vol. 15, No. 3, March 1989, pp. 305-313.

[Gehani 84] Gehani, N., *Ada: Concurrent Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[Ghezzi 82] Ghezzi, C. and Jazayeri, M., *Programming Language Concepts*. John Wiley and Sons, Inc., 1982, 327 pp.

[Glass 82] Glass, R.L., *Modern Programming Practices*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Grafton 85] Grafton, R.B., and Ichikawa, T., eds. *IEEE Computer*, Special Issue of Visual Programming, Vol. 18, No. 8, August 1985.

[Guarna 89] Guarna, V.A., Gannon, D., Jablonowski, D., Malong, A.D., and Gaur, Y., "Faust: An Integrated Environment for Parallel Programming," *IEEE Software*, Vol. 6, July 1989, pp. 20-27.

[Haibt 59] Haibt, L.M., "A Program to Draw Multi-Level Flow Charts," *Proceedings of the Western Joint Computer Conference*, San Francisco, CA, Vol. 15, March 1959, pp. 131-137.

[Hayes 89] Hayes, F., and Baran, N., "A Guide to GUIs," *BYTE*, Vol. 14, No. 7, July 1989, pp. 250-257.

[Isoda 87] Isoda, S., Shimomura, T., and Ono, Y., "VIPS: A Visual Debugger," *IEEE Software*, Vol. 4, No. 3, May 1987, pp. 8-19.

[Lanovaz 88] Lanovaz, D.A., *Godel: A Prototype Prolog Programming Environment*, Department of Computing Science, University of Alberta, M. Sc. thesis, Oct. 1988, 121 pp.

[Leintz 80] Leintz, B.P., and Swanson, E.B., *Software Maintenance Management*, Addison-Wesley, Reading, Mass., 1980.

[London 85] London, R.L. and Duisberg, R.A., "Animating Programs using Smalltalk," *IEEE Computer*, Special Issue of Visual Programming, Vol. 18, No. 8, August 1985.

[Model 79] Model, M.I., *Monitoring System Behavior In a Complex Computational Environment.* Xerox PARC CSL-79-1, Palo Alto, January 1979. 179 pp.

[Moriconi 85] Moriconi, M., and Hare, D.F., "Visualizing Program Designs Through PegaSys," *IEEE Computer*, Vol. 18, No. 8, August 1985, pp. 72-85.

[Myers 80] Myers, B.A., *Displaying Data Structures for Interactive Debugging*. Xerox PARC CSL-80-7, Palo Alto, June 1980, 97 pp.

[Myers 83] Myers, B.A., "Incense: A System for Displaying Data Structures," *Computer Graphics*, Vol. 17, No. 3, July 1983.

[Myers 85] Myers, B.A., "What are Visual Programming, Programming by Example, and Program Visualization?," Department of Computer Science, University of Toronto, October 1985.

[Panel Session 83] Panel Session on Debugging Methodology, *Proceedings of Symposium on High Level Debugging*, Asilomar, California, 1983.

[Reiss 84] Reiss, S.P., "Graphical Program Developement with PECAN Development Systems," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices*, May 1984, pp. 30-41.

[Reiss 86] Reiss, S.P., Golin, E.J. and Rubin, R.V., "Prototyping Visual Languages with the GARDEN System," *Proceedings of the IEEE Computer Society Workshop on Visual Languages*, June 1986, pp. 81-90.

[Seymour 89] Seymour, J., "The GUI: An Interface You Won't Outgrow," *PC Magazine*, Vol. 8, No. 15, September 1989, pp. 97-109.

[Sherman 84] Sherman, M., and Kratzer, A., "View-3 and Ada: Tools for Building Systems with Many Tasks," in *Proceeding Washington Ada Symposium*, Washington, D.C., March 1984.

[Sherwood 86] Sherwood, B.A., and Sherwood, J.N., "CMU Tutor: An Integrated Programming Environment for Advanced-Function Workstations," *Proceedings of the IBM Academic Information Systems University AEP Conference*, Vol. IV, April 1986, pp. 29-37.

[Szafron 86a] Szafron, D., and Wilkerson, B., "GUIDE: Preliminary Users' Manual," Technical Report TR 86-05, Department of Computing Science, University of Alberta, March 1986.

[Szafron 86b] Szafron, D., Wilkerson, B., Tam, S., Lanovaz, D.A. and Lew, E., *GUIDE: A Graphical Programming Environment for Procedural Languages*, Preliminary Report, Department of Computing Science, University of Alberta, August 1986.

[Tandem 85] *DEBUG Manual for Tandem Non-Stop Systems*, Part No. 82598 A00, Tandem Computers Inc., CA, March 1985.

[Teitelman 77] Teitelman, W., *A Display Oriented Programmer's Assistant*. Xerox PARC CSL-77-3, Palo Alto, March 1977, 30 pp.

[Teitelman 81] Teitelman, W. and Masinter, L., "The Interlisp Programming Environment," *IEEE Computer*, Vol. 14, No. 4, April 1981, pp. 25-34.

[Teitelman 85] Teitelman, W., "A Tour Through Cedar," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, March 1985, pp. 285-302.

[van Tassel 74] van Tassel, D., *Program Style, Design, Efficiency, Debugging and Testing*. Prentice-Hall, Englewood Cliffs, NJ, 1974, 256 pp.

[Yarwood 77] Yarwood, E., *Toward Program Illustration*. University of Toronto Computer Systems Research Group Technical Report CSRG-84, Octobert 1977 (M. Sc. Thesis).